A comparative survey of recent natural language interfaces for databases

Affolter, Katrin ; Stockinger, Kurt ; Bernstein, Abraham

Abstract: Over the last few years, natural language interfaces (NLI) for databases have gained significant traction both in academia and industry. These systems use very different approaches as described in recent survey papers. However, these systems have not been systematically compared against a set of benchmark questions in order to rigorously evaluate their functionalities and expressive power. In this paper, we give an overview over 24 recently developed NLIs for databases. Each of the systems is evaluated using a curated list of ten sample questions to show their strengths and weaknesses. We categorize the NLIs into four groups based on the methodology they are using: keyword-, pattern-, parsing- and grammar-based NLI. Overall, we learned that keyword-based systems are enough to answer simple questions. To solve more complex questions involving subqueries, the system needs to apply some sort of parsing to identify structural dependencies. Grammar-based systems are overall the most powerful ones, but are highly dependent on their manually designed rules. In addition to providing a systematic analysis of the major systems, we derive lessons learned that are vital for designing NLIs that can answer a wide range of user questions.

DOI: https://doi.org/10.1007/s00778-019-00567-8

The VLDB Journal
https://doi.org/10.1007/s00778-019-00567-8

REGULAR PAPER

A comparative survey of recent natural language interfaces for databases

Katrin Affolter[1] · Kurt Stockinger[1] · Abraham Bernstein[2]

Received: 18 September 2018 / Revised: 17 July 2019 / Accepted: 17 August 2019

Abstract
Over the last few years, natural language interfaces (NLI) for databases have gained significant traction both in academia and industry. These systems use very different approaches as described in recent survey papers. However, these systems have not been systematically compared against a set of benchmark questions in order to rigorously evaluate their functionalities and expressive power. In this paper, we give an overview over 24 recently developed NLIs for databases. Each of the systems is evaluated using a curated list of ten sample questions to show their strengths and weaknesses. We categorize the NLIs into four groups based on the methodology they are using: keyword-, pattern-, parsing- and grammar-based NLI. Overall, we learned that keyword-based systems are enough to answer simple questions. To solve more complex questions involving subqueries, the system needs to apply some sort of parsing to identify structural dependencies. Grammar-based systems are overall the most powerful ones, but are highly dependent on their manually designed rules. In addition to providing a systematic analysis of the major systems, we derive lessons learned that are vital for designing NLIs that can answer a wide range of user questions.

Keywords Natural language interfaces · Query processing · Survey

1 Introduction

Living in a digital age, the global amount of data generated is increasing rapidly. A good part of this data is (semi-)structured and stored in some kind of database, which is usually accessed using query languages such as SQL or SPARQL. Structured query languages, however, are difficult to understand for non-experts. Even though SQL was initially developed to be used by business people, reality shows that even technically skilled users often experience problems putting together correct queries [8], because the user is required to know the exact schema of the databases, the roles of various entities in the query and the precise join paths to be followed. Non-technical (or casual) users are usually overwhelmed by the technical hurdles of formal query languages.

One often-mentioned approach to facilitate database querying even for casual users is the use of *natural language interfaces* (NLI) for databases. These allow users to access information stored in databases by typing questions expressed in natural language [24]. Some NLIs restrict the use of the natural language to a sub-language of the domain or to a natural language restricted (and sometimes controlled) by grammatical constraints. For example, SODA [6] is based on English keywords, ATHENA [48] handles full sentences in English, and Ginseng [5] strictly guides the user to a correct full sentence in English. If users want to find all movies with the actor Brad Pitt, for instance, the input question in the different systems could be as follows:

SODA: `movie Brad Pitt`
ATHENA: `Show me all movies with the actor Brad Pitt.`
Ginseng: `What are the movies with the actor Brad Pitt?`

Depending on the database schema, the corresponding SQL statement might be as follows:

✉ Kurt Stockinger
Kurt.Stockinger@zhaw.ch

[1] Zurich University of Applied Sciences, Winterthur, Switzerland

[2] University of Zurich, Zurich, Switzerland

Published online: 28 August 2019

```
SELECT m.title FROM Movie m
  JOIN Starring s ON s.movieId
 = m.id
  JOIN Actor a ON a.actorId
 = s.actorId
  JOIN Person p ON p.id
= a.actorId
 WHERE p.FirstName = "Brad"
   AND p.LastName = "Pitt"
```

As we can see in this example, the users of NLIs do not have to know the underlying structure or query language to formulate the question in English.

Critiques of NLIs often highlight that natural language is claimed to be too verbose and too ambiguous. If it were possible to identify the different types of linguistic problems, then the system could support the user better, for example, with a clarification dialog. This would not only help the NLI to translate the natural language question into a formal query language, but would also assist the users in formulating correct queries. For example, if the users ask a question like '*all movies from Columbia Pictures or 20th Century Fox*,' the system should ask for clarification of the ambiguous token '*Columbia*' which can be either a location or a movie company (as part of the bi-gram '*Columbia Pictures*'). If the users choose movie company, then the system could directly suggest that '*20th Century Fox*' is also a movie company and not the concept for films from 1900 until 1999.

Additionally, natural language is not only ambiguous on word level, but there can be also multiple interpretations of the meaning of a sentence. In the example input question '*all cinemas in cities with parking lots*' the constraint '*with parking lots*' can either refer to the cinemas or to the cities. Both interpretations are grammatically correct, but probably the intent is to find cinemas with a parking area. In some cases, like '*all cinemas in cities with love seats*' there is only one interpretation valid on the data set: only cinemas can have a constraint on '*love seats*,' even if it would be grammatically correct for cities. Therefore, an NLI should first check if multiple interpretations exist. Next, it should verify if they can be applied on the dataset. If there are still multiple interpretations left, the users should have the possibility to choose the most relevant one.

The above-mentioned examples are only some of the linguistic problems that NLIs must deal with. Furthermore, users are not perfect and tend to make mistakes. These range from spelling errors to the use of colloquial language, which includes syntactically ill-formed input. These examples do, however, highlight that precisely understanding the expressiveness of questions that an NLI is able to interpret is of paramount importance both for developers and users of NLIs. To address this need, this paper makes the following **contributions**:

- We provide an overview of recent NLI systems comparing and analyzing them based on their expressive power.
- Existing papers often use different data sets and evaluation metrics based on precision and recall, while others perform user studies to evaluate the system (see Sect. 5 for details). Given this heterogeneity of evaluations, it is very hard to directly compare these systems. Hence, we propose a set of sample questions of increasing complexity as well as an associated domain model aiming at testing the expressive power of NLIs.
- The paper serves as a guide for researchers and practitioners, who want to give natural language access to their databases.

To help the reader understand the differences between current NLIs, in Sect. 2, we first describe a sample world, which we use as an example running consistently through the paper, in order to evaluate and benchmark the systems. This is different to previous surveys on NLIs [20,37,43,44,49], where the systems are only summarized or categorized. Based on our sample world, we introduce ten representative sample questions of increasing expressive complexity and illustrate their representativeness using question-answering corpora. These questions are used to compare the NLIs and highlight the various difficulties the NLIs face. In Sect. 3, we give a brief overview of the most important technology for natural language processing and discuss the limitations of our evaluation in Sect. 4. The sample world and the questions are then used in Sect. 5 to explain the process of translating from natural language questions to a formal query language for each NLI surveyed. We provide a systematic analysis of the major systems used in academia (Sect. 6.1) as well as three major commercial systems (Sect. 6.2). In Sect. 7, we discuss newer developments of NLIs based on machine learning. Note that these systems are not the main focus of the paper but serve more as a brief insight into new research avenues. Finally, we derive lessons learned that are vital for designing NLIs that can answer a wide range of user questions (Sect. 8).

Note that the detailed evaluation of the systems in Sect. 6 omits the approaches based on machine learning. The rationale for this is twofold. First, the functionality of the machine learning-based approaches is heavily dependent on the training data. Most papers present a system trained with some dataset and then show the capabilities of the system in the context of that training. We found no exploration of the general capabilities or robustness of a given approach when varying the input data. Hence, but is difficult to say if these systems could cover all requirements when given suitable training data or not. Second, little is known how domain-dependent those systems are on the training data. Usually, they require a lot of training examples making the comparison to the other systems—that mainly require some metadata—difficult. We, hence, concluded that the categorization of the

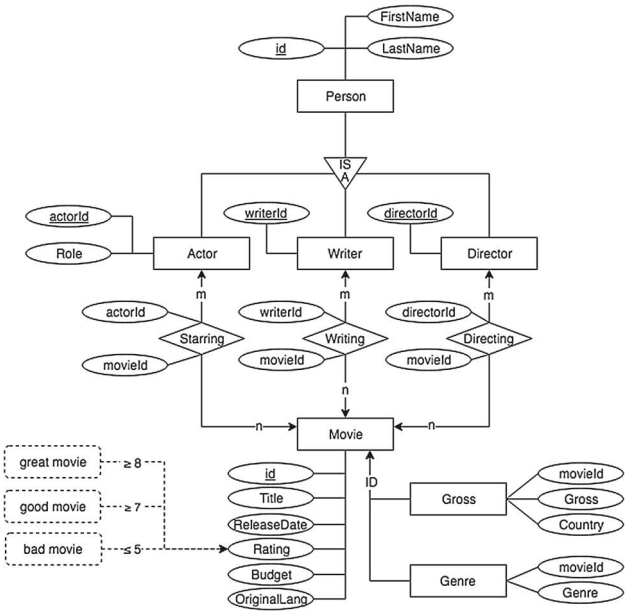

Fig. 1 Ontology of the sample world 'movie database'

capability of these systems is an open problem that needs to be addressed in its own survey.

2 Foundation: a sample world

In this section, we present a small sample world that serves as the basis for analyzing different NLIs. We first introduce the database schema (Sect. 2.1) and afterward discuss ten input questions of increasing complexity that pose different challenges to NLIs (Sect. 2.2). Finally, we will perform an analysis of different question-answering corpora to better understand what types of questions real users pose and how we can map them to our ten input questions. Our analysis in Sect. 2.3 indicates that our ten sample questions represent a large range of questions typically queried in question-answering systems.

2.1 Database ontology

The sample world is a small movie database inspired by IMDB[1] and extended with hierarchical relationships and semantic concepts. Figure 1 visualizes the ontology of the sample world, which consists of movies, persons and their relationships employing an entity-relation diagram. A person can be an actor, a writer and/or a director. For movies, information about starring actors, writers and directors is stored. Each movie has a title, a release date, a rating (float), a budget (bigint) and the original language. It can also have

[1] https://www.imdb.com/.

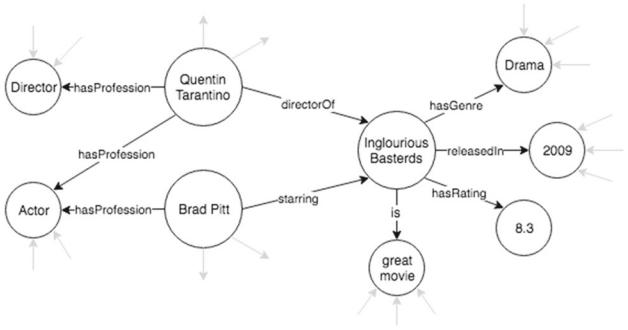

Fig. 2 Part of the knowledge graph representing the sample world 'movie database'

multiple genres and multiple gross profits for different countries. Furthermore, there are three concepts—*bad*, *good* and *great* movie—defined, all depending on the rating of the movies. Elements with solid lines correspond to the schema of the underlying database. The entries in the database will be called **base data**. The concepts and the schema are the **metadata**. To improve performance, many NLIs implement inverted indices over literal values such as strings (both from the base and metadata). Elements with dotted lines correspond to (possibly derived) concepts, which are only defined in the ontology.

Another possible representation of the data would be as a knowledge graph. The core concepts of a knowledge graph are the entities and their relationships. Figure 2 visualizes a part of the sample world as a knowledge graph. Concepts can be directly included in the knowledge graph. For example, the movie '*Inglourious Basterds*' has a link to the concept '*great movie*.'

2.2 Input questions

To explain the different NLIs and highlight their strengths and weaknesses, input questions are necessary. NLIs are designed to help non-tech users to formulate formal queries using natural language. The design of our questions is inspired by Turing Award Winner Jim Gray who designed 20 challenging questions of increasing complexity that were used as the yard stick for evaluating large-scale astrophysics databases [55]. This methodology of designing questions of increasing complexity also served as the basis for the evaluation of SODA [6] and the ten sample questions of this paper.

Therefore, we designed nine questions based on the operators of SQL and SPARQL: **J**oins, **F**ilters (**s**tring, **r**ange, **d**ate or **n**egation), **A**ggregations, **O**rdering, **U**nion and **S**ubqueries.

Furthermore, we added a question which is based on a **c**oncept (e.g., '*great movie*'). **C**oncepts are not part of SQL and SPARQL, but a common addition of NLIs. Table 1 shows

Table 1 Ten sample input questions based on SQL/SPARQL operators that are answerable on the sample world. (**J**oin; **F**ilter (**s**tring, **r**ange, **d**ate or **n**egation); **A**ggregation; **O**rdering; **U**nion; **S**ubquery; **C**oncept)

#	Natural language question	Challenges
Q1	Who is the director of 'Inglourious Basterds'?	J, F(s)
Q2	All movies with a rating higher than 9.	J, F(r)
Q3	All movies starring Brad Pitt from 2000 until 2010.	J, F(d)
Q4	Which movie has grossed most?	J, O
Q5	Show me all drama and comedy movies.	J, U
Q6	List all great movies.	C
Q7	What was the best movie of each genre?	J, A
Q8	List all non-Japanese horror movies.	J, F(n)
Q9	All movies with rating higher than the rating of 'Sin City'.	J, S
Q10	All movies with the same genres as 'Sin City'.	J, 2xS

those ten full sentence questions in English, which can be applied on the sample world (ordered roughly by difficulty).

The queries were designed in such a way that they cover a wide range of SQL functionality (technical challenge) as well as linguistic variations (semantic challenge). We will now analyze each of these ten queries in more detail and describe the major challenges for the underlying system to solve them.

The first question (Q1) is a join over different tables (Person, Director, Directing and Movie) with an ISA-relationship between the tables Person and Director. Moreover, the query has a filter on the attribute Movie.Title, which has to be equal to '*Inglourious Basterds*.' Therefore, the system faces three different challenges: (a) identify the bridge table Directing to link the tables Director and Movie, (b) identify the hierarchical structure (ISA-relationship) between Director and Person and (c) identify '*Inglourious Basterds*' as a filter phrase for Movie.Title.

The second question (Q2) is based on a single table (Movie) with a range filter. The challenge for the NLIs is to translate '*higher than*' into a comparison operator '*greater than*.'

The third question (Q3) is a join over four tables (Person, Actor, Starring and Movie) and includes two filters: (a) a filter on the attribute Person.FirstName and Person.LastName and (b) a two-sided date range filter on the attribute Movie.ReleaseDate. The challenge in this query (compared to the previous ones) is the date range filter. The system needs to detect that '*from 2000 until 2010*' refers to a range filter and that the numbers need to be translated into the dates *2000-01-01* and *2010-12-31*.

The fourth question (Q4) is a join over two tables (Movie and Gross). In addition, an aggregation on the attribute Gross.Gross and grouping on the attribute Movie.id or ordering the result based on Gross.Gross is needed. For both approaches, an aggregation to a single result (indicated by the keyword '*most*') is requested.

The fifth question (Q5) is a join over two tables (Movie and Genre). The query can either be interpreted as '*movies that have both genres*' (intersection) or '*movie with at least one of those genres*' (union). The expected answer is based on the union interpretation, which can be solved with two filters that are concatenated with an OR on the attribute Genre.Genre.

The sixth question (Q6) needs the definition of concepts. In the sample world, the concept '*great movie*' is defined as a movie with a rating greater or equal 8. If the system is capable of concepts, then it needs to detect the concept and translate it accordingly to the definition.

The seventh question (Q7) is a join over two tables (Movie and Genre) with an aggregation. The challenges are to (a) identify the grouping by the attribute Genre.Genre and (b) translate the token '*best*' to a maximum aggregation on the attribute Movie.Rating.

The eighth question (Q8) is a join over two tables (Movie and Genre) with a negation on the attribute Movie.OriginalLang and a filter on the attribute Genre.Genre. The challenge in this question is to identify the negation '*non-Japanese*.' Another possible input question with a negation over a larger movie database, would be '*All actors without an Oscar*.' Here again, the challenge is to identify '*without*' as a keyword for the negation.

The ninth question (Q9) is based on a single table (Movie) and includes a subquery. The challenge in this question is to divide it in two steps: first select the rating of the movie '*Sin City*' and then use this SQL statement as a subquery to compare with the ranking of every other movie in the database.

The tenth question (Q10) is a join over two tables (Movie and Genre). One possible solution would include two not exist: the first one verifies for each movie that there exist no other genres as the genres of '*Sin City*.' The second one verifies for each movie that it has no genre, which '*Sin City*' does not have. For example, the movie '*Sin City*' has the genre '*Thriller*,' the movie '*Mission: Impossible*' has the genres

'*Thriller*' and '*Action*.' The first `not exist` will check if '*Mission: Impossible*' has the genre '*Thriller*' from '*Sin City*' which is true. The second `not exist` checks if '*Sin City*' has the genres '*Thriller*' and '*Action*' (from '*Mission: Impossible*'), which is false.

2.3 Question analysis

In this section, we perform an analysis comparing our ten sample input questions with the two well-known question-answering corpora Yahoo! QA Corpus L6[2] (more than 4 million questions within a user community) and GeoData250 [56] (250 questions against a database). We also summarize the findings of Bonifati et al. [7] and compare them to our input questions. The goal of the analysis is to better understand what types of questions users pose and how representative our sample input questions are (i.e., establish some external validity of their representativeness).

For the Yahoo! Corpus, we decided to only look into the labeled subset of movie questions since our sample world is based on movies. Out of these questions related to movies, we extracted a random sample set of 100 questions. We used all the GeoData250 Questions. We labeled each of those 350 questions from both sources with `Q1` to `Q10` based on what poses the challenge in answering them.

For example, the GeoData250 question '*give me the cities in virginia?*' is labeled with `Q1`, because the challenge of this question is to identify the right filter ('*virginia*'). The question '*what is a good movie to go see this weekend?*' includes a time range and is therefore labeled as `Q6`. For the Yahoo! Corpus, we also made some assumptions, for example, the question '*what is your favorite tom hanks movie?*' is interpreted as '*give me the best ranked tom hanks movie*' and labeled with `Q4`. Furthermore, if a question could have multiple labels, the label of the more difficult (higher number) question is chosen. For example, the sample question '*can anyone tell a good action movie to watch?*' is labeled with `Q6` because it requires handling of a concept ('*good movie*') and not `Q1` because it uses a filter ('*action movie*'). If the question cannot be labeled with one of the input questions, we label it with `x`.[3] For example, the question '*i want to make a girl mine but she is more beautiful than me. what can i do now?*' has nothing to do with movies.

As shown in Fig. 3, more than 40% of the Yahoo! questions are based on filtering only, i.e., corresponding to question `Q1`. For example, the question '*what movie had "wonderful world" by sam cooke at the beginning?*' has filters for the song '*wonderful world*' and a join on movie. About 30% of the questions are labeled with `x` which are off-topic questions. There are no questions labeled with `Q2`; this means that there are no questions with a numerical range. This can be explained by the composition of the corpus itself, which is a collection of questions from users to users. If users ask about the ranking of a movie, they ask something like '*what is your favorite movie?*' and not something similar to `Q2`.

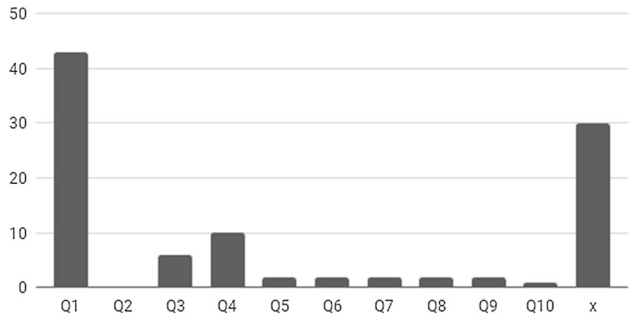

Fig. 3 Mapping of 100 sample questions of Yahoo! QA Corpus L6 (movie) to our ten sample world questions

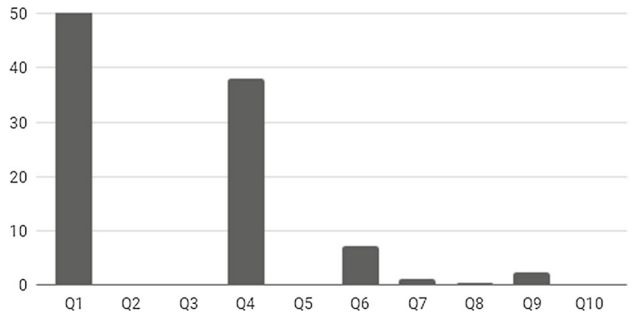

Fig. 4 Mapping of the GeoQuery250 questions (in percentage) to our ten sample world questions

In Fig. 4, we can see the distribution of question types for the GeoData250 corpus. About 88% of the questions are labeled with `Q1` or `Q4`. There are three concepts ('*population density*,' '*major city*' and '*major river*') used in the corpus that occur in roughly 8% of the questions, 7% of which are labeled with `Q6`. There are no numerical range questions (`Q2`) and no date questions (`Q3`). The latter can be explained by the dataset not including any dates. There are also no unions (`Q5`) and no questions with multiple subqueries (`Q10`).

Bonifati et al. [7] investigated a large corpus of query logs from different SPARQL endpoints. The query log files are from seven different data sources from various domains. One part of the analysis was done by counting the SPARQL keywords used in the queries. Over all domains, 88% of the queries are `Select`-queries and 40% have `Filter`. Furthermore, they found huge differences between different domains. For example, the use of `Filter` ranges from 61% (LinkedGeoData) to 3% (OpenBioMed) or less. This implies that the distribution of the usage for the question types is domain-dependent. Nevertheless, our ten sample questions

[2] https://webscope.sandbox.yahoo.com/catalog.php?datatype=l.

[3] This label is only needed for the Yahoo! questions, where the topic is not about movies, even though they were labeled as 'movie' by Yahoo!

are fully covered in the query log analyzed by Bonifati et al. [7].

Our analysis indicates that our ten sample questions represent a large range of questions typically queried in question-answering systems. Note that open-ended questions—the ones we labeled as `x` in the Yahoo! Corpus—are not covered by our analysis.

3 Background: natural language processing technologies

In this section, we will discuss the most commonly used *natural language processing* (NLP) technologies that are relevant for NLIs to databases. In particular, we give a short overview on handling of stop words and synonyms, tokenization, part of speech tagging, stemming, lemmatization and parsing.

3.1 Stop word

The term *stop word* refers to the most common words in a language. There is no clear definition or official list of stop words. Depending on the given purpose, any group of words can be chosen as stop words. For example, search engines mostly remove function words, like *the*, *is*, *at* and others. Punctuation marks like dot, comma and semicolon are often included in the stop word list. For NLIs, stop words can contain invaluable information about the relationship between different tokens. For example, in the question '*What was the best movie of each genre?*' (`Q7`) the stop words '*of each*' imply an aggregation on the successive token '*genre*.' On the other hand, stop words should not be used for lookups in the inverted indexes. In the question '*Who is the director of "Inglourious Basterds"?*' (`Q1`), the stop word '*of*' would return a partial match for a lot of movie titles, which are not related to the movie '*Inglourious Basterds*.' Therefore, the NLIs should identify stop words, but not remove them because they can be useful for certain computations.

3.2 Synonymy

The difficulty of synonymy is that a simple lookup or matching is not enough. For example, the question '*All movies* `starring` *Brad Pitt from 2000 until 2010.*' (`Q3`) could also be phrased as '*All movies* `playing` *Brad Pitt from 2000 until 2010.*' The answer should be the same, but because in the sample world no element is named '*playing*,' a lookup would not find an answer. Therefore, it is necessary that the system takes synonyms into account. A possible solution is the use of a translation dictionary. Usually, such a dictionary is based on DBpedia [34] and/or WordNet [42].

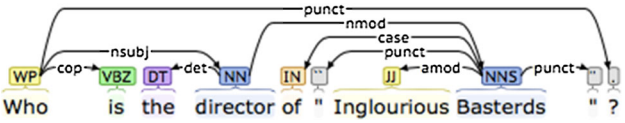

Fig. 5 PoS tags and dependency tree for the sample question `Q1` using Stanford CoreNLP. *PoS tags*: `WP` = wh-pronoun; `VBZ` = verb, third person singular present; `DT` = determiner; `NN` = noun, singular or mass; `JJ` = adjective; `NNS` = noun, plural *Dependencies*: `punct` = punctuation; `nsubj` = nominal subject; `cop` = copula; `det` = determiner; `nmod` = nominal modifier; `case` = case marking; `amod` = adjectival modifier

3.3 Tokenization

Tokenization is used to split an input question into a list of tokens. It can be as simple as a separator on whitespace, but more often it is based on multiple rules (e.g., with regular expressions) or done with ML algorithms. Simple whitespace splitting tokenizers are often not good enough if the input question includes punctuation marks. For example, all input questions (`Q1-10`) end either with a question mark or a period. If the punctuation mark is not separated from the last word, the NLI would have to search for a match for the token '*Basterds"?*' (`Q1`) instead of '*Basterds*.' Without other processing, the NLI will not find any full matches. Depending on the task to solve, part of the tokenization process can be splitted on punctuation marks or deleting them. Either way, there are some scenarios to think about. For example, decimals should neither be split on the punctuation mark nor should they be removed. Consider the following example '*All movies with a rating higher than 7.5*' (similar to `Q2`). If you remove the dot between 7 and 5, the result would be completely different. Also other NLP technologies could be dependent on punctuation marks, for example, dependency trees.

3.4 Part of speech tagging

A *part of speech* (PoS) is a category of words with similar grammatical properties. Almost all languages have the PoS tags *noun* and *verb*. PoS tagging is the process of annotating each token in a text with the corresponding PoS tag (see Fig. 5). The tagging is based both on the token itself and its context. Therefore, it is necessary to first tokenize the text and identify end-of-sentence punctuation. More advanced NLP technologies use the information produced by the PoS tagger. For example, both lemmatization and dependency tree parsing of Stanford CoreNLP [40] have the requirement for PoS tagging.

3.5 Stemming/lemmatization

The goal of both stemming and lemmatization is to reduce inflectional and derivationally related forms of a word to a common base form.

Stemming reduces related words to the same stem (root form of the word) by removing different endings of the words. To achieve that most stemming algorithms refer to a crude heuristic process that chops off the ends of words. The most common algorithm for stemming in English is the Porter's algorithm [46]. It is based on simple rules that are applied to the longest suffix. For example, there is a rule '`ies` → `i`' which means that the suffix '`ies`' will be reduced to '`i`.' This is needed for words like '*ponies*' which are reduced to '*poni*.' In addition, there is a rule '`y` → `i`' which ensures that 'pony' is also reduced to '*poni*.' In the sample world, stemming can be used to ensure that the words '*directors*,' '*director*,' '*directing*' and '*directed*' can be used to find the table `Director`, because they are all reduced to the same stem '*direct*.' The disadvantage of stemming is that the generated stem not only consists of words with a similar meaning. For example, the adjective '*direct*' would be reduced to the same stem as '*director*,' but the meaning differs. An example question could be '*Which movie has a direct interaction scene between Brad Pitt and Jessica Alba?*,' where the word '*direct*' has nothing to do with the director of the movie. In general, stemming increases recall but harms precision.

Lemmatization removes inflectional endings and returns the lemma, which is either the base form of the word or the dictionary form. To achieve that lemmatization algorithms usually use a vocabulary and morphological analysis of the words. For example, '*directors*' and '*director*' would both have the lemma '*director*' but '*directing*' and '*directed*' would lead to the verb '*(to) direct*.' Lemmatization is normally used together with PoS tagging, which leads to the distinction between the verb '*direct*' and the adjective '*direct*.' Another example would be the question '*Who wrote "Inglourious Basterds"?*' where lemmatization can translate the irregular verb '*wrote*' into '*write*.' In contrast to stemming, lemmatization can be improved by using context.

3.6 Parsing

Parsing is the process of analyzing the grammatical structures (syntax) of a sentence. Usually, the parser is based on context-free grammar. There are two main directions of how to look at the syntax: the first main direction, dependency syntax (Fig. 5), looks at the syntactic structures as relations between words. The other main direction, constituency syntax (Fig. 6), analyzes not only the words but also more complex entities (constituents). The information generated through parsing is traditionally stored as a syntax tree. NLIs can use the information on relations found in the syntax tree to generate more accurate queries.

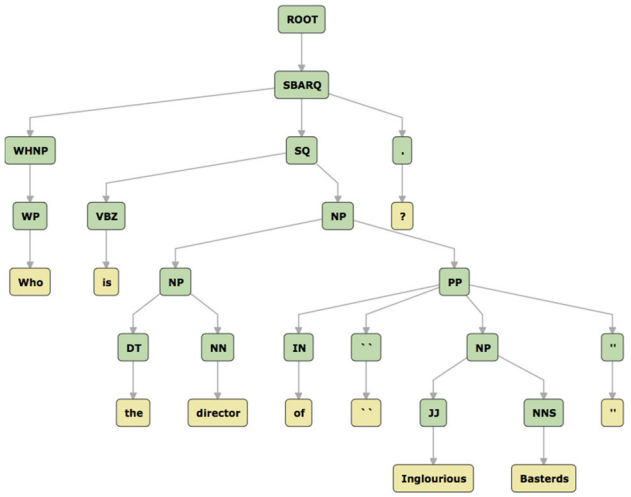

Fig. 6 Constituency tree for the sample question `Q1` using Stanford CoreNLP

4 Limitations

This survey's evaluation focuses on the ten sample questions introduced in Sect. 2.2. Those sample questions are based on the operators of the formal query languages SQL and SPARQL. This leads to the following limitations of the evaluation.

Limitation 1—Theoretical Our evaluation is theoretical and only based on the papers. A few systems have online demos (e.g., SPARKLIS [19]), but others—especially older ones—are not available any more. Therefore, to handle all systems equally, we based our evaluation fully on the papers of the systems.

Our approach is based on the assumption that the papers contain enough information in order to reproduce the reported experimental results. Hence, when analyzing the papers against our ten queries of increasing complexity, we explicitly checked if these papers contain hints about being able to answer these specific queries, if their architecture is in principle able to answer them or if the papers discuss explicit constraints about not being able to answer the respective queries. In short, we performed a very systematic approach of checking each of the 24 papers based on these ten queries. One potential limitation of our approach is that the actual implementation of the paper could differ from the description in the paper, and hence some of the results might be incorrect.

Limitation 2—Computational performance We completely ignored the computational performance of the systems. This limitation is based on the different focuses of the systems. For example, Walter et al. [58] (BELA) propose an approach to increase the speed and computational efficiency. Zheng et al. [69] (NLQ/A) and Zenz et al. [68] (QUICK) pro-

pose algorithms to optimize the number of user interactions for solving ambiguity problems.

Limitation 3—Accuracy In our evaluation, we ignored the accuracy given in the papers. There are multiple reasons for this decision: (a) Depending on the focus of the system, not all papers include evaluations based on accuracy. (b) Different usages of the same metrics, for example, Saha et al. [48] (ATHENA) only consider questions where the system could calculate an answer, but Damljanovic et al. [12] (FREyA) evaluate the system against the same dataset but based on needed user interactions. (c) The number of questions in different datasets can be highly different, for example, GeoData250 consists of 250 questions, Task 2 of QALD[4]-4 only of 25. (d) Even if two systems are using the same dataset, two system can preform different preprocessing, for example, Kaufmann et al. [31] (QUERIX) reduces the GeoData880 to syntactic patterns instead of all questions. All those factors make it impossible to directly compare the systems based on the given evaluation metrics in the paper.

5 Recently developed NLIs

In this section, we will focus on more recent NLIs starting from 2005. We will not discuss about older systems like BASEBALL [21], LUNAR [64], RENDEZVOUS [9], LADDER [47], Chat-80 [61], ASK [57] and JANUS [62], which are often quoted in the research field of NLIs. We will systematically analyze 24 recently developed systems in Sects. 5.1–5.4 based on the sample world introduced in Sect. 2. The main goal is to highlight strengths and weaknesses of the different approaches based on a particular data model and particular questions to be able to directly compare the systems. The explanation is based on the information describing the systems found in the papers. Finally, in Sect. 6.1, we will evaluate the systems against the sample questions and give an overall interpretation of the system.

There are different ways to classify NLIs. In this survey, we divide the NLIs into four main groups based on the technical approach they use:

1. **Keyword-based systems**
   The core of these systems is the lookup step, where the systems try to match the given keywords against an inverted index of the base and metadata. These systems cannot answer aggregation queries like question Q7 '*What was the best movie of each genre?*' The main advantage of this approach is the simplicity and adaptability.
2. **Pattern-based systems**
   These systems extend the keyword-based systems with NLP technologies to handle more than keywords and also add natural language patterns. The patterns can be domain-independent or domain-dependent. An example for a domain-independent pattern would be allowing aggregations with the words '*by*' or '*how many*.' A domain-dependent pattern could for example be a concept like '*great movie*.'
3. **Parsing-based systems**
   These systems parse the input question and use the generated information about the structure of the question to understand the grammatical structure. The parse tree contains a lot of information about single tokens, but also about how tokens can be grouped together to form phrases. The main advantage of this approach is that the semantic meaning can be mapped to certain production rules (query generation).
4. **Grammar-based systems**
   The core of these systems is a set of rules (grammar) that define the questions a user can ask the system. The main advantage of this approach is that the system can give users natural language suggestions during typing their questions. Each question that is formalized this way can be answered by the system.

Table 2 gives an overview of the most representative NLIs of these four categories that we will discuss in Sects. 5.1–5.4. The table also shows which kind of query languages the systems support (e.g., SQL) as well as which NLP technologies are used. In summary, our approach of systematically evaluating these systems on a sample world with queries of increasing complexity enables a better comparison of the different approaches.

In the next subsections, we will systematically analyze the systems in more detail.

5.1 Keyword-based systems

The core of keyword-based NLIs is their lookup step. In this step, the system tries to match the given keywords against an inverted index of the base and metadata. To identify keywords in the input question, some systems are using stop word removal (e.g., NLP-Reduce [32]), others are expecting only keywords from the users as input (e.g., SODA [6]).

Most questions are easily formulated with keywords. However, there are some cases where keywords are not enough to express the intention of the users. For example, for the question '*What was the best movie of each genre?*' (Q7), the 'keyword-only version' would be something like '*best movie genre*,' which is more likely to be interpreted as '*the genre of the best movie*.' If the users would write the question like '*best movie by genre*,' a keyword-based NLI would try to lookup the token '*by*' in the base and metadata or classify '*by*' as a stop word and ignore it.

[4] http://qald.aksw.org/.

Table 2 Categorization of the NLIs and the used NLP technologies

			Stop word	Synonym	PoS	Stem / Lemma	NER	Dep. Parser	Const. Parser	SQL	SPARQL	Other
Keyword	**SODA**	*2012*	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗
	NLP-Reduce	*2007*	✗	✓	✗	✓	✗	✗	✗	✓	✗	✓
	Précis	*2008*	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗
	QUICK	*2009*	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗
	QUEST	*2013*	?	?	?	?	✗	✗	✗	✓	✗	✗
	SINA	*2015*	✓	✗	✗	✓	✗	✗	✗	✗	✓	✗
	Aqqu	*2015*	?	✓	✓	✓	✓	✗	✗	✗	✓	✗
Pattern	**NLQ/A**	*2017*	✓	✓	✗	✓	✗	✗	✗	✗	✓	✗
	QuestIO	*2008*	✗	?	✓	✓	✗	✗	✗	✗	✓	✗
Parsing	**ATHENA**	*2016*	✗	✓	✓	✓	✓	✓	✗	✓	✗	✗
	Querix	*2006*	✗	✓	✓	?	✗	✗	✓	✗	✓	✗
	FREyA	*2010*	?	✓	✓	?	✗	✗	✓	✗	✓	✗
	BELA	*2012*	✗	✓	✓	?	✗	✗	✓	✗	✓	✗
	USI Answers	*2013*	?	✓	✓	✓	✓	✓	✗	✓	✓	✓
	NaLIX	*2005*	✗	✓	✓	?	✗	✗	✓	✗	✗	✓
	NaLIR	*2014*	✗	✓	✓	?	✗	✗	✓	✓	✗	✗
	BioSmart	*2017*	?	?	✓	?	✗	✗	✓	✓	✗	✓
Grammar	**TR Discover**	*2015*	?	▲	✗	✗	✗	✗	✗	✓	✓	✗
	Ginseng	*2005*	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗
	SQUALL	*2014*	✗	✓	✓	✗	✗	✗	✓	✗	✓	✗
	MEANS	*2015*	✓	✓	✓	✓	✓	✗	✗	✗	✓	✗
	AskNow	*2016*	✗	✓	✓	✓	✓	✗	✗	✗	✓	✗
	SPARKLIS	*2017*	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗
	GFMed	*2017*	?	?	?	?	?	?	?	✗	✓	✗

✓, using; ▲, partly using; ✗, not using; ?, not documented

In the following, we will summarize seven keyword-based NLI. We decided to describe SODA [6]—as the first system—in depth, because it can solve the most of our sample input questions in this category (see Sect. 2.2). SODA is an NLI that expects only keywords from the user and can handle aggregations by using specific non-natural language templates. Afterward, the other systems are summarized, and we highlight the difference between them to SODA and each other.

5.1.1 SODA (Search Over DAta warehouse)

SODA [6] is a system that provides a keyword-based NLI for relational databases with some extensions in the direction of a pattern-based system. The base data consists of the relational database. The metadata can include multiple ontologies, which are handled like natural language patterns. For example, domain-specific ontologies with concepts (like the concept '*great movie*' in the sample world) or DBpedia to identify homonyms and synonyms. SODA uses both inverted indexes (base and metadata) as the basis for finding query matches in the data. The key innovation of SODA is that it provides the possibility to define metadata patterns which specify conceptual models. The concept '*good movie*' could depend on various variables not only on the rating, but for example, also on the number of ratings. The users can then apply this concept to their input questions, for example, they could search for '*all great movie*' (`Q6`) without having to specify what a great movie is.

Assuming the users want to know the director of the movie '*Inglourious Basterds*' (`Q1`), the input question for SODA could be: '*director Inglourious Basterds*.'

SODA uses five steps to translate this keyword-based input question into a SQL query. The first step is the lookup: it checks the keywords against the inverted indexes over the database and provides all the nodes in the metadata graph where these keywords are found. For the input question `Q1`, this means that the keyword '*director*' can be found in the inverted index of the meta data, either the table name `Director` or to the attribute name `Director.directorId` and `Directing.directorId` (Fig. 7: red). The keyword '*Inglourious Basterds*' is only found in the inverted index of the base data as a value of the attribute `Movie.Title` (Fig. 7: green). This leads to three different solution sets for the next steps: {`Directing.directorId`, `Movie.Title`},

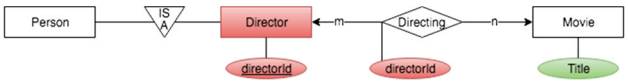

Fig. 7 Nodes in the metadata graph corresponding to the keywords '*director*' (red) and '*Inglourious Basterds*' (green) found during the lookup step of SODA (color figure online)

{Director.directorId, Movie.Title} and {Director, Movie.Title}.

The second step is to assign a score to each solution of the lookup step. SODA uses a simple heuristic method, for example, in-domain solutions receive a higher score. For the input question, the solution {Director, Movie.Title} receives the highest score, because the table name Director is a full match and not only a fuzzy match like in directorId. Afterward, only the best n solutions are provided to the next step.

The third step identifies which tables are used for each of the solutions provided by the previous step. Also, the relationships and inheritance structures between those tables are discovered in this step. For the best solution of the input question, the tables Director and Movie correspond to the different entry points. An entry point is a node in the metadata graph. The table Director is a child of the table Person (ISA-relationship). Therefore, SODA includes the table Per-son in the solution. To link the table Movie to the other two tables, it is necessary to add more tables to the solution. The closest link is through the table Directing (see Fig. 7), and therefore this table is included.

The fourth step collects the filters. There are two types of filters which are collected. The first one are filters in the input question like '*Inglourious Basterds*.' The second one are filter conditions that occur during traversing the metadata graph like the concept '*great movie*.'

The fifth and last step generates a *reasonable* and *executable* SQL query from the information collected in the previous steps. A *reasonable* SQL query is a query which considers foreign keys and inheritance patterns in the schema. An *executable* SQL query is a query that can be executed on the underlying database.

The strengths of SODA are the use of metadata patterns and domain ontologies, which allow one to define concepts and include domain-specific knowledge. In addition, the inclusion of external sources like DBpedia for homonyms and synonyms is beneficial for finding meaningful results. Furthermore, SODA is designed to evolve and thus improve over time based on user feedback.

The weaknesses of SODA are that it uses simple word recognition for comparison operators. For example, to retrieve all movies with a rating greater than 9, the input question needs to be written like '*rating* > *9*' (Q2). Moreover, SODA uses a very strict syntax for aggregation operators. For example, to retrieve the number of movies per year, the input question needs to be written like '*select count (movie) group by (year)*.' These patterns are useful, but are not in natural language. Furthermore, there is no lemmatization, stemming or any other preprocessing of the input question which can lead to a problem with words that are used in plural. For example the input question '*all movies*' would not detect the table Movie but the input question 'all movie' would display the expected result.

Blunschi et al. [6] suggest extending SODA to handle temporal aspects of the data warehouse (e.g., bi-temporal historization). They also pointed out that the GUI of SODA should be improved so that the users are engaged in selecting and ranking the different results. Furthermore, the user feedback provided by SODA is currently very basic and needs to be improved.

5.1.2 NLP-reduce

NLP-Reduce [32] uses a few simple NLP technologies to 'reduce' the input tokens before the lookup in the *Knowledge Base* (KB) based on RDF. The system takes the input question, reduces it to keywords, translates it into SPARQL to query the KB and then returns the result to the user.

In contrast to SODA, NLP-Reduce uses not only synonyms but also two other NLP technologies: (a) stop words and punctuation marks removal and (b) stemming. With the removal of stop words and punctuation marks, NLP-Reduce is able to answer some fragment and full sentence questions. NLP-Reduce still cannot answer questions with aggregations like '*What was the best movie of each genre?*' (Q7), because it will remove the token '*of each*' as a stop word. Furthermore, stemming helps the user to formulate questions like '*all movies*' which is more natural than '*all movie*' for SODA.

After reducing the input question, NLP-Reduce has similar steps to SODA: (1) search for triples in the RDF graph (similar to base and metadata), where at least one of the question tokens occurs and rank the found triples, (2) search for properties that can be joined with the triples found in the previous step by the remaining question tokens, (3) search for datatype property values that match the remaining question tokens and (4) generate corresponding SPARQL query.

Compared to SODA, the strength of NLP-Reduce is the reduction in the input question such that non-keyword input questions can be answered. Besides the overall weakness of keyword-based NLIs, NLP-Reduce is not able to answer comparison questions like Q2.

5.1.3 Précis

Précis [51] is a keyword-based NLI for relational databases, which supports multiple terms combined through the operators AND, OR and NOT. For example, input question '*Show me all drama and comedy movies.*' (Q5) would be formulated as '*"drama" OR "comedy"*.' The answer is an entire multi-

relation database, which is a logical subset of the original database.

First, Précis transforms the input question into *disjunct normal form* (DNF). Afterward, each term in the DNF is looked up in the inverted index of the base data. This is different to SODA, where the inverted index includes the metadata. If a term cannot be found, the next steps are not executed. The third step creates the schema of the logical database subset, which represents the answer of the input question. This includes the identification of the necessary join paths.

The strength of Précis is the ability to use brackets, `AND`, `OR` and `NOT` to define the input question. However, the weaknesses are that this again composes a logical query language, although a simpler one. Furthermore, it can only solve Boolean questions, and the input question can only consist of terms which are located in the base data and not in the metadata. For example, the input question '*Who is the director of "Inglourious Basterds"?*' (`Q1`) cannot directly be solved because '*director*' is the name of a table and therefore part of the metadata. There is a mechanism included that adds more information to the answer (e.g., the actors, directors etc., to a movie), but then the user would have to search for the director in the answer.

5.1.4 QUICK (QUery Intent Constructor for Keywords)

QUICK [68] is an NLI that adds the expressiveness of semantic queries to the convenience of keyword-based search. To achieve this, the users start with a keyword question and then are guided through the process of incremental refinement steps to select the question's intention. The system provides the user with an interface that shows the semantic queries as graphs as well as textual form.

In a first step, QUICK takes the keywords of the input question and compares them against the KB. Each possible interpretation corresponds to a semantic query. For example, the input question '*Brad Pitt*' can either mean '*movies where Brad Pitt played in*,' '*movies directed by Brad Pitt*' or '*movies written by Brad Pitt*' (see Fig. 1). In the next step, the system provides the users with the information in such a way that they can select the semantic query which will answer their question. To do so, QUICK provides the users with possible interpretations of each keyword to select from. This is done with a graph as well as a textual form of the semantic query. The textual form is a translation of the SQL query into natural language based on templates. Furthermore, the system orders the keywords in such a way that the user interactions are as few as possible. When the users select the desired semantic query, QUICK executes it and displays the results in the user interface.

The strength of QUICK is the user interaction interface with the optimization for minimal user interaction during the semantic query selection. The weakness of QUICK is that it is limited to acyclic conjunctions of triple patterns.

5.1.5 QUEST (QUEry generator for STructured sources)

QUEST [4] is a keyword-based NLI to translate input questions into SQL. It combines semantic and statistical ML techniques for the translation.

The first step is to determine how the keywords in the input question correspond to elements of the database (lookup). In contrast to SODA, QUEST uses two *Hidden Markov Models* (HMM) to choose the relevant elements (ranking). The first HMM is a set of heuristic rules. The second HMM is trained with user feedback. The next step is to identify the possible join paths to connect all the relevant elements from the previous step. QUEST selects the most informative join paths (similar to SODA's step 4). The most informative join paths are those that contain tuples in the database. In the third step, QUEST decides which combination of keyword mapping and join path most likely represents the semantics the users had in mind when formulating the keyword question.

The strength of QUEST is the combination of user feedback and a set of heuristic rules during the ranking. This allows the system to learn from the users over time. A weakness of QUEST is that it is not able to handle concepts such as '*good movie*.'

5.1.6 SINA

SINA [50] is a keyword-based NLI that transforms natural language input questions into conjunctive SPARQL queries. It uses a *Hidden Markov Model* to determine the most suitable resource for a given input question from different datasets.

In the first step, SINA reduces the input question to keywords (similar to NLP-Reduce), by using tokenization, lemmatization and stop word removal. In the next step, the keywords are grouped into segments, with respect to the available resources. For example, the keywords '*Inglourious*' and '*Basterds*' would be grouped into one segment based on the match for '*Inglorious Basterds*.' In the third step, the relevant resources are retrieved based on string matching between the segments and the RDF label of the resource. In the following step, the best subset of resources for the given input question is determined (ranking). The fifth step, SINA constructs the SPARQL query using the graph-structure of the database. Finally, the results, retrieved by evaluating the generated SPARQL query, are shown to the users.

The biggest weakness of SINA is that it can only translate into conjunctive SPARQL queries, which reduce the number of answerable questions.

5.1.7 Aqqu

Aqqu [2] is an NLI which uses templates to identify possible relations between keywords. At the end of the translation process, ML is used to rank the possible solutions.

To translate the input question into SPARQL, first the entities from the KB that match (possibly overlapping) parts of the input question are identified. The possible parts are identified by using PoS tags. For example, single token parts must be a noun (`NN`) and proper nouns (`NNP`) are not allowed to be split (e.g., '*Brad Pitt*'). In the next step, Aqqu uses three different templates which define the general relationship between the keywords. Afterward, Aqqu tries to identify the corresponding relationship. This can either be done with the help of the input question (verbs and adjectives), or with the help of ML which for example can identify abstract relationship like '*born* → *birth date*.' The last step is the ranking which is solved with ML. The best result is achieved by using a binary random forest classifier.

The strength of Aqqu is the identification of abstract relationships. The weakness is the limitation of a keyword-based NLI.

5.2 Pattern-based systems

Pattern-based NLIs are an extension of keyword-based systems with natural language patterns to answer more complex questions like concepts (`Q6`) or aggregations (`Q7`). For example, the question '*What was the best movie of each genre?*' (`Q7`) cannot be formulated with keywords only. It needs at least some linking phrase between '*best movie*' and '*genre*,' which indicates the aggregation. This could be done with the non-keyword token (trigger word) '*by*' for the aggregation, which will indicate that the right side includes the keywords for the `group by`-clause and the left side the keywords for the `select`-clause. The difficulty with trigger words is to find every possible synonym allowed by natural language. For example, an aggregation could be implied with the word '*by*' but also '*of each*' (compare `Q7`).

In the following, we summarize two pattern-based NLIs. We decided to describe NLQ/A [69] in depth, because it is based on the idea that the errors made by NLP technologies are not worth the gain of information. Instead, the system is highly dependent on the users' input to solve ambiguity problems, and therefore it focuses on the optimization of the user interaction.

5.2.1 NLQ/A

NLQ/A [69] is an NLI to query a knowledge graph. The system is based on a new approach without NLP technologies like parsers or PoS taggers. The idea being that the errors made by these technologies are not worth the gain of information. For example, a parse tree helps for certain questions like subqueries (e.g., `Q9`), but if the parse tree is wrong, the system will fail to translate even simpler questions. Instead, NLQ/A lets the users resolve all ambiguity problems, also those which could be solved with PoS tagging or parse trees. To avoid needing too many interaction steps, NLQ/A provides an efficient greedy approach for the interaction process.

Assuming the users want to know the director of the movie '*Inglourious Basterds*' (`Q1`), the input question could be: '*Who is the director of "Inglourious Basterds"?*.'

NLQ/A use four steps to answer the input question. The first step is to detect the phrases of the input question. In general, the phrases can be categorized into two types: *independent* and *dependent* phrases. *Independent* phrases are identified with a phrase dictionary. The dictionary consists of variables, aggregations, operators, modifiers and quantifier phrases. To detect *dependent* phrases, most stop words are removed (simplified input question). Some types of words like prepositions are still needed and therefore kept. Next `1:n`-grams are generated. Phrases starting with prepositions are discarded. After stop word removal, the input question `Q1` would become '*director of Inglourious Basterds*.' If `n` is set to 2, the extracted phrases would be: {'*director*,' '*director of*,' '*Inglourious*,' '*Inglourious Basterds*,' '*Basterds*.'} Next, the phrases are extended according to a synonym dictionary. For example if there is a phrase '*starring*,' it would be extended with the phrase '*playing*.' Those extended phrases are mapped to the knowledge graph based on the string similarity (edit distance). For one extended phrase, there can be multiple candidate mappings.

The next step takes the candidate mappings and tries to find the true meaning of the input question with the help of the users. To reduce the amount of interactions for the user, a *phrase dependency graph* (PDG) is proposed. The PDG consists of two parts: (PDG1) a graph where each node represents a phrase, two phrases are connected if they share at least one common token and (PDG2) a subgraph of the knowledge graph consisting of the candidates where each node represents a candidate, two nodes are connected if they are adjacent in the knowledge graph. The two parts are connected with edges, representing the mapping between phrases and candidates (see Fig. 8).

In the third step, the users get involved to solve the ambiguity given in the PDG. In order to reduce the necessary user interactions, the NLI tries to find those edges which resolve the most ambiguities (similar to the idea of QUICK).

The last step takes the selected candidates and tries to connect them into one graph. The connected graph will include the answer to the question. Groups of already connected candidates in the PDG2 are called query fragments. In Fig. 8, the candidates '*director-Of*' and '*Inglourious Basterds*' are one query fragment. For each query fragment, the system tries to find the path with the highest similarity to the sim-

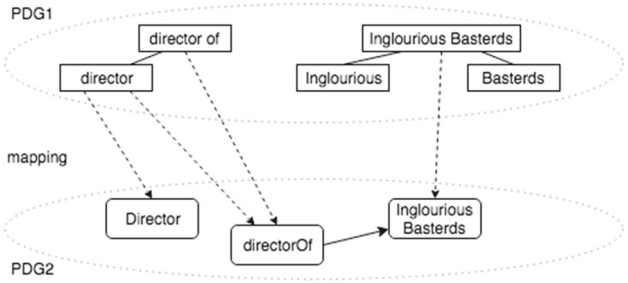

Fig. 8 Phrase dependency graph (PDG) for the input question '*Who is the director of "Inglourious Basterds"?*.' (PDG1: input question; PDG2: knowledge graph)

Fig. 9 Answer graph generated for the input question '*Who is the director of "Inglourious Basterds"?*' based on the selected candidates (blue) (color figure online)

plified input question. For the input question `Q1`, if the users select '*Director*' as candidate in step 3, the system would find the path as shown in Fig. 9. '*Inglourious Basterds*' is also a candidate, but not selected by the users because there is no ambiguity to solve.

The strengths of this NLI are the simplicity and the efficient user interaction process. The simplicity allows easy adaption on new knowledge graphs and together with the user interaction process it overcomes the difficulties of ambiguity.

The weakness of this system is that usually more than one user interaction is needed to resolve ambiguities, in the experiments the average number of interactions was three [69].

5.2.2 QuestIO (QUESTion-based Interface to Ontologies)

QuestIO [11] is an NLI to query ontologies using unconstrained natural language. It automatically extracts human-understandable lexicalization from the ontology. Therefore, the quality of the semantic information in the ontology has to be very high to contain enough human-understandable labels and/or descriptions. For example, the attribute `Movie.Release-Date` would be extracted as '*Release Date*,' which is a human-understandable label. In contrast, the attribute `Movie.OriginalLang` would result in '*Original Lang*,' where the token '*Lang*' is a shortened version for '*Language*' and is not human-understandable.

QuestIO translates the input question with three steps: In the first step, the *key concept identification tool* identifies all tokens which refer to mentions of ontology resources such as instances, classes, properties or property values. This is similar to the dependent phrases of NLQ/A. In the next step, the context collector identifies patterns (e.g., key phrases like '*how many*') in the remaining tokens that help the system to understand the query (similar to independent phrases of NLQ/A). The last step identifies relationships between the ontology resources collected during the previous steps and formulates the corresponding formal query. After executing the query, it will be sent to the *result formatter* to display the result in an user-friendly manner.

The automatic extraction of semantic information out of the ontology is both a strength and a weakness of QuestIO. It is highly dependent on the development of the human-understandable labels and descriptions, without them QuestIO will not be able to match the input questions to the automatic extracted information.

5.3 Parsing-based systems

Parsing-based NLIs are going a step further than previously discussed systems: they parse the input question and use the information generated about the structure of the question to understand the grammatical structure. For example, the grammatical structure can be used to identify the dependencies given by the trigger word '*by*' in a question. This is needed for long-range dependencies which cannot be caught with simple natural language patterns. Furthermore, the dependency parser can help to handle the difficulty of verboseness. For example, the nominal modifier (`nmod`) could be used to identify aggregations.

In the following, we summarize eight parsing-based NLIs. We decided to describe ATHENA [48] in depth, because it can answer the most of the sample input questions. Furthermore, ATHENA uses the most NLP technologies, and the authors describe all the steps in depth. Afterward, the other systems are summarized, and we highlight the delta to ATHENA and previous systems.

5.3.1 ATHENA

ATHENA [48] is an ontology-driven NLI for relational databases, which handles full sentences in English as the input question. For ATHENA, ontology-driven means that it is based on the information of a given ontology and needs mapping between an ontology and a relational database. A set of synonyms can be associated with each ontology element. During the translation of an input question into a SQL query, ATHENA uses an intermediate query language before subsequently translating it into SQL.

Assuming the users want to know the director of the movie '*Inglourious Basterds*' (`Q1`), the input question for ATHENA could be: '*Who is the director of "Inglourious Basterds"?*.'

ATHENA uses four steps to translate a full sentence input question into a SQL query. In the first step, the ontology

Fig. 10 Matches found by ATHENA with the ontology evidence annotator for the input question '*Who is the director of "Inglourious Basterds"?*' (color figure online)

evidence annotator is used, which maps the input to a set of ontology elements. There are five types of possible matches:

a. *metadata*: Finding a match in the inverted index for the metadata (and the associated set of synonyms). Longer matches over the input question are preferred if there are multiple matches.
b. *translation index*: The translation index is an extension of the inverted index over the base data, which is enriched with variations for person and company names. For example, for the person name '*Brad Pitt*,' there would also be an entry '*B. Pitt*.'
c. *time range expressions*: Finding all time ranges like '*from 2000 until 2010*' (Q5) with the TIMEX annotator. Those time ranges are then matched to the ontology properties with the corresponding data type.
d. *numeric expressions*: Finding all tokens that include numeric quantities with the Stanford Numeric Expressions annotator. The numeric quantities can be either in the form of numbers (e.g., 9) or in text form (e.g., nine). Those numeric expressions are then matched to the ontology properties with the corresponding datatype.
e. *dependencies*: Annotating dependencies between tokens in the input question. For example, in the input question Q1, there is a dependency between the tokens '*director*' and '*Inglourious Basterds*' indicated by the token '*of*.'

For the input question Q1, the metadata annotation will detect three different matches for '*director*,' namely the table name `Director` and the attribute name `Director.directorId` and `Directing.directorId` (Fig. 10: red). The translation index will find a match for the bi-gram '*Inglourious Basterds*,' corresponding to the attribute `Movie.Title` (Fig. 10: green).

The next step generates a ranked list of interpretations. An interpretation is a set of ontology elements provided by the previous step. If n ontology elements exist for one token, there will also be n different interpretations, one for each ontology element. For the given input question, there are three different interpretations possible: {`Directing.directorId`, `Movie.Title`}, {`Director.directorId`, `Movie.Title`} and {`Director`, `Movie.Title`}. Each interpretation is represented by a set of interpretation trees. An *interpretation tree* (iTree) is a subtree of the ontology. Each iTree must satisfy:

a. *evidence cover*: All tokens, which were annotated in the previous step, need to be covered.
b. *weak connectedness*: All concepts need to be at least weakly connected through an undirected path and each property must be connected to its corresponding concept. For the first interpretation this means that `Director` and `Movie` need to be connected, for example, via the relation `Directing`. The attribute `Title` needs to be connected with the corresponding concept (in this case the table) `Movie`.
c. *inheritance constraint*: No ontology element is allowed to inherit from its child concepts. For example, the ontology element `Person` is not allowed to inherit `Role` of `Actor`. The other direction is allowed, such that `Actor` inherits `FirstName` and `LastName` from `Person`.
d. *relationship constraint*: All relationships given in the input question are included, not depending on the direction of the path. For example, the tree tokens '*movies*,' '*starring*' and '*Brad Pitt*' (Q5) imply a relationship constraint between the ontology element `Movie`, `Starring` and `Person`. Those three ontology elements need to be connected. Accordingly, in this example, the ontology element `Actor` needs to be included.

For the purpose of ranking the different interpretations, ATHENA generates one single iTree. It can consist of a union of multiple iTrees or a single iTree. Figure 11 shows a possible iTree for the interpretation {`Director`, `Movie.Title`}, which is extended with the ontology element `Directing` and `Movie`. After this step, for each interpretation only one iTree is left.

The third step uses the ranked list of interpretations to generate an intermediate query in the *Ontology Query Language* (OQL). OQL was specifically developed for ATHENA to be an intermediate language between the input question and SQL and is able to express queries that include aggregations, unions and single nested subqueries. The structure of an OQL query is similar to SQL and is generated as follows:

a. `from` *clause*: Specifies all concepts found in the ontology along with their aliases. The aliases are needed, for example, if a concept occurs multiple times. For example, the input question '*Show me all drama and comedy movies.*' (Q4) would point to `Genre` in the ontology twice: once for the token '*drama*' and once for '*comedy*.' Therefore, two aliases are needed to distinguish between them.
b. `group by` *clause*: The `group by` clause is triggered by the word '*by*' and only tokens annotated with metadata in step 1.a are considered. For example, the input question '*What was the best movie by genre?*' (modified Q7). To identify the dependencies between dependent and dependee (illustrated by the '*by*'), the Stanford Dependency Parser is used.

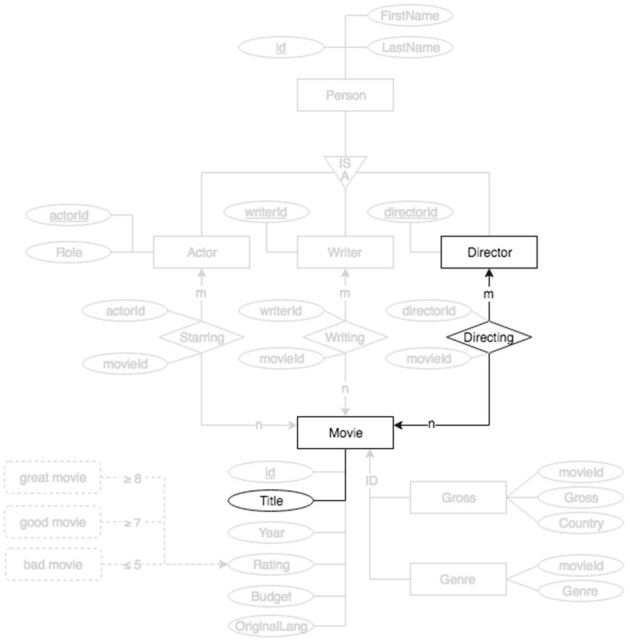

Fig. 11 Possible interpretation tree (black) for the input question '*Who is the director of "Inglourious Basterds"?*' (Q1)

c. `select` *clause*: There are two possible types: *aggregation* and *display* properties. The *aggregation* properties depend on the `group by` clause. The default aggregation function is `sum`. For the (modified) input question Q7, ATHENA would detect a `group by` clause because the '*by genre*' needs an aggregation function. Assuming ATHENA can translate '*best movie*' to mean '*best ranked movie*,' it would apply the aggregation function `max` on `Movie.Rating`. If there are no aggregations, ATHENA uses the tokens annotated with metadata as *display* properties, which are shown to the user.
d. `order by` *clause*: Properties used in the `order by` clause are indicated by tokens like '*least*,' '*most*,' '*ordered by*,' '*top*' and others. For example, the input question '*Which movie has grossed most?*' (Q3) would trigger an `order by` clause for `Movie.Gross` because of the trigger word '*most*.'
e. `where` *clause*: Tokens annotated with the translation index, time range or numerical expression are used in the `where` clause to filter the result (e.g., the tokens '*Inglourious Basterds*'). If the filter is applied on an aggregation, a `having` clause is generated instead of the `where` clause.

The final step translates the OQL query into a SQL query, where each attribute and join condition is either *concrete* or *virtual*. *Concrete* means that a direct mapping between ontology and the database (e.g., '*director*') exist. *Virtual* implies that a (complex) relationship between the ontology elements and the database (e.g., '*great movie*') exists. Furthermore, the result of the best ranked interpretation is directly displayed, but the users will see the other interpretations as well. All the top `n` interpretations that ATHENA has found are translated back into full sentences in English for the users, so that the users can choose the best fitting one.

The strengths of ATHENA are the ontology as an abstraction of the relational database and the natural language explanation for each interpretation of the input question. The translation index contains not only synonyms but also semantic variants for certain types of values like persons and company names. Furthermore, ATHENA can handle single-level nesting in the input question. An example input question could be '*All movies with rating higher than the rating of "Sin City".*' (Q9).

One of the weaknesses of ATHENA is that neither negation (Q8) nor multiple elements in the `group by` clause (e.g., '*What was the best movie by year and genre?*') are supported.

Saha et al. [48] suggest extending ATHENA to handle more than single-level nesting. Furthermore, they suggest enabling a possibility to answer follow-up questions using the context of the previous questions.

5.3.2 Querix

Querix[5] [31] allows users to enter questions in natural language to query an ontology. If the system identifies any ambiguities in the input question, it asks the user for clarification in a dialog. Querix uses a syntax tree to extract the sequence of words from the main word categories: noun (`N`), verb (`V`), preposition (`P`), `wh`-pronoun (`Q`, e.g., what, where, when, etc.,) and conjunction (`C`). This sequence is called query skeleton. The query skeleton is used to enrich nouns and verbs and to identify `subject-property-object` patterns in the query.

In contrast to ATHENA, which uses a lot of different tools and technologies, Querix only uses the information of the query skeleton (parse tree) and the synonyms (for both the input question and the ontology) to translate the input question into SPARQL. For translating into SPARQL, Querix uses three components: *query analyzer*, *matching center* and *query generator*. The *query analyzer* handles two tasks: (1) It applies the Stanford Parser on the input question to generate a syntax tree, from which Querix extracts the query skeleton. For example, the query skeleton '`Q-V-N-P-N`' is extracted from the input question (Q1) as '*Who* (`Q`) *is* (`V`) *the director* (`N`) *of* (`P`) *"Inglourious Basterds"* (`N`) *?*'. (2) It enriches all nouns and verbs with synonyms provided by WordNet.

[5] The name is based on the druid Getafix who is consulted by Asterix (in the same named comic) whenever anything strange occurs.

The *matching center* is the core component of Querix: (1) It tries to match the query skeleton with a small set of heuristic patterns. Those patterns are used to basically identify `subject-property-object` patterns in the input question. (2) It searches for matches between nouns and verbs of the input question with the resources in the ontology (including synonyms). (3) It tries to match the results of the two previous steps. The *query generator* then composes SPARQL queries from the joined triplets delivered by the last step of the matching center. If there are several different solutions with the highest cost score, Querix will consult the user by showing a menu from which the user can choose the intended meaning.

The simplicity of Querix is both a strength and a weakness: it is simple to use and completely portable, but this simplicity also reduces the number of questions that can be answered since they have to adhere to a predefined syntax.

5.3.3 FREyA (Feedback, Refinement and Extended vocabularY Aggregation)

FREyA [12] is based on QuestIO. It allows users to enter queries in any form in English. It generates a syntactic parse tree in order to identify the answer type. The translation process starts with a lookup, annotating query terms with ontology concepts by using heuristic rules. If there are ambiguous annotations, the user will be engaged in a clarification dialogue. The user's selections are saved and used for training the system in order to improve its performance over time. In the end, the system generates a SPARQL query.

The strength of this system is the user interaction, which not only supports the users to find the right answer, but also improves FREyA over time. Furthermore, FREyA performs an answer type identification, which leads to more precise answers. The weakness is that only a few of the questions could be answered without any clarification dialogs. Furthermore, the system cannot answer negations.

5.3.4 BELA

BELA [58] is an NLI with a layered approach. This means, that at each layer the best hypothesis is determined. If the confidence for a particular interpretation is `1` and the SPARQL query generated by it produces an answer with at least one result, the translation process is stopped and the answer is returned to the user. Only for `ASK`-questions (which have yes/no answers), the process continues until the confidence of the interpretations start to differ, then a threshold of 0.9 is applied and an empty result (which equals a *no*-answer) is also accepted.

Similar to Querix, BELA parses the input question and produces a set of query templates, which mirror the semantic structure. The next step is a lookup in the index, including Wikipedia redirects (this corresponds to the translation index of ATHENA). The first lookup is without fuzzy matching, if no result can be found, a threshold of 0.95 is applied for a matching with normalized Levenshtein distance. The third lookup enables the usage of synonyms, hypernyms and hyponyms from WordNet. The last lookup (and step) uses *Explicit Semantic Analysis*, which can be used to relate expressions like '*playing*' to concepts like '*actor*.'

Compared to other systems, BELA not only focuses on solving the translation task, but also reduces the computation time, which increases the user-friendliness.

5.3.5 USI Answers

USI Answers [59] is an NLI for semi-structured industry data. It offers natural language access to the large bodies of data that are used in the planning and delivery of services by Siemens Energy. The database consists of multiple databases, for example domain-specific ontologies, different relational databases and SPARQL endpoints. These can be either internal or external knowledge (e.g., DBpedia). Furthermore, the users demanded to be able to use not only natural language questions and formal query language constructs but also keyword questions or a mixture of these.

In contrast to the previous parsing-based NLIs, USI Answers cannot rely on the parse tree, because of the different types of possible input (e.g., keyword questions). Still, the first step includes various NLP technologies, such as lemmatization, PoS tagging, named entity recognition and dependency parsing. To handle domain-specific input terms, there is a list of syntax rules to revalidate the entity information after the NLP technologies were applied. Furthermore, the question focus (i.e., referenced entity object) is identified by applying 12 syntactic rules. Enriched with this information, the next step aims to identify and resolve the different concepts that may be in the input question (lookup). Next, USI Answers detects and prevalidates the different relations and objects found in the previous step. The fourth step generates different question interpretations, including how concepts and instances are connected to each other. Afterward, the different interpretations are validated and ranked. This is done with a learned model, based on user-defined views. The last step constructs the final query in the representative query language.

One strength of USI Answers is the ability to query different database resources. Furthermore, the users are free to choose their preferred type of input questions.

5.3.6 NaLIX (Natural Language Interface to XML)

NaLIX [38] is an interactive NLI for querying an XML database with XQuery. The interaction is based on guiding the users to pose questions that the system can handle

by providing meaningful feedback and helpful rephrasing suggestions. Furthermore, NaLIX provides templates and question history to the users. It preserves the users prior search efforts and provides the users with a quick starting point when they create new questions.

Similar to Querix, NaLIX mostly uses the parse tree for the translation of an input question into XQuery. After MiniPar[6] is used to parse the input question, NaLIX identifies the phrases in the parse tree of the input question that can be mapped to XQuery components and classifies them. In the next step, NaLIX validates the classified parse tree from the previous step: it checks if it knows how to translate the classified parse tree into XQuery and if all attribute names and values can be found in the database. The last step translates the classified parse tree into an appropriate XQuery expression if possible. During both the classification and the validation of the parse tree, information about errors (e.g., unknown phrases and invalid parse trees) is collected to report to the users for clarification.

The strength of NaLIX is the ability to solve difficult questions, which can include subqueries, by using and adjusting the parse tree. On the other hand, the reliance on a parse tree is also a weakness, because the system can only answer questions that are parseable.

5.3.7 NaLIR (Natural Language Interface for Relational databases)

NaLIR [35,36] is a further development of NaLIX. Instead of translating into XQuery to query XML databases, NaLIR translates the input question into SQL and queries relational databases. NaLIR returns not only the result to the user, but also a rephrased version of the input question based on the translation.

The basic idea remains the same: parse the input question using the Stanford Parser and map the parse tree to SQL (instead of XQuery). The steps are similar with some modifications: In the step of mapping phrases of the input question parse tree to SQL components, the users are asked for clarification if there are ambiguities. The next step is not a validation anymore, but an adjustment of the parse tree in such a way that it is valid. The candidate interpretations produced by this adjusted and valid parse tree are delivered to the users to select the correct interpretation. The last step translates the adjusted and valid parse tree which the user has selected into SQL.

The strength of NaLIR is the interaction with the user, which improved further compared to NaLIX. The weakness remains: it is highly dependent on the parse tree.

[6] https://gate.ac.uk/releases/gate-7.0-build4195-ALL/doc/tao/splitch17.html.

5.3.8 BioSmart

BioSmart [27] uses a syntactic classification of the input question to translate the natural language question into a declarative language such as SQL. The system divides the input questions into three query types (iterative, conditional or imperative) using the parse tree of the input question and compare it to parse tree templates for each query type. For example, a imperative query consists of a *verb* (VB) followed by a *object* (NP). A more expressive and therefore complex input question can be built by nesting simple query types arbitrarily.

Similar to Querix, BioSmart uses the Stanford parser to parse the input question. The system then tries to map the resulting parse tree to predefined questions or to one of the query templates (query type identification). As mentioned, it is possible that a question consists of several of these templates to capture the meaning of the question. Then, the tables and possible joins that are needed to compute the query are identified. Afterward, the templates are transformed into a logical query using the information about the tables and joins.

Compared to other NLI, the strength of BioSmart is the possibility to query arbitrary databases. The weakness of BioSmart is the mapping to the three query types: if the system cannot match the input question to those query types, it is not able to answer the question.

5.4 Grammar-based systems

Grammar-based NLIs use a different approach. The core of these systems is a set of rules (grammar) that define the questions that can be understood and answered by the system. Using those rules, the system is able to give the users suggestions on how to complete their questions during typing. This supports the users to write understandable questions and gives the users insight into the type of questions that can be asked. The disadvantage of these systems is that they are highly domain-dependent: the rules need to be written by hand.

In the following, we summarize seven grammar-based NLIs. We decided to describe TR Discover [52] in depth, because it is well-described in the paper such that we can provide examples for the whole process, including the grammar rules. Furthermore, it uses the rules to guide and help the users during formulating their questions. Afterward, other grammar-based systems are summarized, and we describe the delta to TR Discover. The differences can be quite significant, however, they all have the same core: a set of rules.

5.4.1 TR Discover

TR Discover [52] is a system providing an NLI which translates the input question in form of an English sentence (or

sentence fragment) into SQL or SPARQL. It is either used for relational databases or ontologies, but does not need an ontology to work for relational databases. During the translation steps, TR Discover uses a *First Order Logic* (FOL) representation as an intermediate language. Furthermore, it provides auto-suggestions based on the user input. There are two types of suggestions: auto-completion and prediction.

TR Discover helps the users in formulating the question through an auto-suggestion feature. For example, assuming the users want to know the director of the movie '*Inglourious Basterds*' (Q1). When the users start typing '*p*,' TR Discover will not only suggest '*person*' but also longer phrases like '*person directing*' (autocomplete). After '*person directing*' is selected (or typed), TR Discover will again suggest phrases, like '*movies*' or even specific movies like '*Inglourious Basterds*' (prediction). For input question Q1, the input could be '*person directing Inglourious Basterds*.'

The suggestions are based upon the relationships and entities in the dataset and use the linguistic constraints encoded in a *feature-based context-free grammar* (FCFG). The grammar consists of grammar rules (G1-3) and lexical entries (L1-2). For the sample world (and the input question Q1), the following rules could be defined:

```
G1: NP → N
G2: NP → NP VP
G3: VP → V NP
L1: N[TYPE=person, NUM=sg, SEM=
    <x.person(x)>] → person
L2: V[TYPE=[person,movie,title],
    SEM=<X x.X(y.directMovie(y,x)>
    , TNS=presp] → directing
```

The suggestions are computed based on the idea of left-corner parsing: given a query segment, it finds all grammar rules whose left corner on the right side matches the left side of the lexical entry of the query segment. Then, all leaf nodes (lexical entries) in the grammar that can be reached by using the adjacent element are found. For example, while typing '*person*' (Q1), the lexical entries L1 and L2 are found and provided to the user.

TR Discover uses three steps to translate the English sentence or fragment of a sentence into a SQL or SPARQL query. The first step parses the input question into a FOL representation. The query parsing uses the FCFG. For the example input, the token '*person*' will be parsed by the lexical entry L1 and the token '*directing*' will be parsed with the lexical entry L2. This leads to the FOL representation:

```
x.person(x) → directMovie(y,x)&
type(y,Movie) & label(y, 'Inglourious
Basterds')
```

How exactly the phrase '*Inglourious Basterds*' is matched to the base data and therefore can be used as part of the lexical

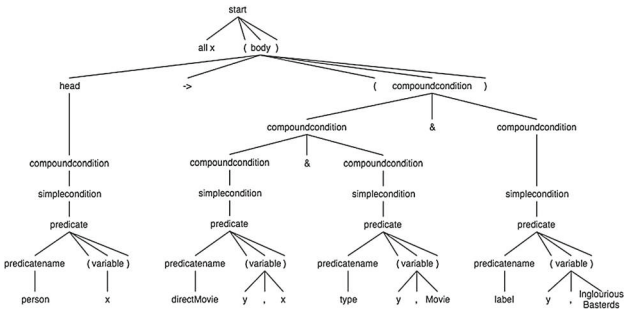

Fig. 12 Parse tree for the FOL representation of the input question '*person directing Inglourious Basterds*'

entry L2 and how it is resolved is not explained by Song et al. [52]. If there are multiple possibilities to parse the input question, the first one is chosen.

The next step is to translate the generated FOL into a parse tree. The FOL parser takes a grammar and the FOL representation from the previous step, and generates a parse tree (Fig. 12) using ANTLER for implementation.

In the third step, an in-order traversal of the parse tree (provided by the previous step) is performed to translate it into an executable SQL or SPARQL query. While traversing the parse tree, the atomic logical conditions and connectors are put on a stack. After traversing, the constraints are popped from the stack to build the correct query constraints. The predicates are mapped to their corresponding attribute names (SQL) or ontology properties (SPARQL).

The strengths of TR Discover are the auto-suggestion and the possibility to translate natural language into different query languages such as SQL and SPARQL, because FOL is used as an intermediate language.

The weaknesses of TR Discover are that quantifiers (e.g., Q3: '*grossed most*') cannot be used, synonyms are not properly handled, and negations only work for SPARQL.

Song et al. [52] suggest extending TR Discover with a ranking system for multiple parses in the first step and to improve the synonym handling. Furthermore, they pointed out the possibility of applying user query logs to improve auto-suggestions.

5.4.2 Ginseng (Guided Input Natural language Search ENGine)

Ginseng [5] is a guided input NLI for ontologies. The system is based on a grammar that describes both the parse rules of the input questions and the query composition elements for the *RDF Data Query Language* (RDQL) queries. The grammar is used to guide the users in formulating questions in English.

In contrast to TR Discover, Ginseng does not use an intermediate representation and therefore the parsing process

translates directly into RDQL. The grammar rules are divided in two categories: *dynamic* and *static* grammar rules. The *dynamic* grammar rules are generated from the OWL ontologies. They include rules for each class, instance, objects property, data type property and synonyms. The *static* grammar rules consist of about 120 mostly empirically constructed domain-independent rules, which provide the basic sentence structures and phrases for input questions. The naming conventions used by Ginseng differ slightly from these used by TR Discover. Ginseng's dynamic rules correspond to TR Discover's lexical rules and Ginseng's static rules consist of both grammar and lexical rules in TR Discover.

The strength of Ginseng are the dynamic rules which are generated from the ontology. This, together with the domain-independent static rules, leads to an easier adaptability compared to systems like TR Discover. The weakness lies in the grammar rules: they need to cover all possible types of questions the users want to ask.

5.4.3 SQUALL (Semantic Query and Update High-Level Language)

SQUALL [17,18] is an NLI for searching and updating an RDF store. It uses the style of Montague grammars (context-free generative grammar) as an intermediate language (similar to TR Discover) to split the translation process in two parts: translating the natural language input question into a logical language and translating the logical language into a query language. Because of that the second part is getting easier: the logical language and the query language share the same semantics and level of detail. The grammar of SQUALL consists of about 120 domain-independent rules.

The translation into the logical form is done in three steps. In the first step, the keywords are recognized (lookup step). The second step is a syntactic analysis based on a descending parser, which is fed with the grammar rules. Afterward, the next step can generate the logical language based on the definition in the grammar. After the translation into the logical language, the translation in to the chosen formal language can be done.

The strength of SQUALL is that it is able to translate any type of input question, including aggregations, negations and subqueries. The weakness of SQUALL is that the users have to know the RDF vocabulary (e.g., classes and properties). For example, the input question `Q1` needs to be formulated as '*Who is the director of* `Inglourious_Basterds`?'

5.4.4 MEANS (MEdical question ANSwering)

MEANS [3] is an NLI that uses a hybrid approach of patterns and ML to identify semantic relations. It is highly domain-dependent and focuses on factual questions expressed by `wh`-pronouns and Boolean questions in a medical subfield targeting the seven medical categories: problem, treatment, test, sign/symptom, drug, food and patient.

To translate the input question into SPARQL, MEANS first classifies the input question into one of ten categories (e.g., factoid, list, definition, etc.). If the question is categorized as a `wh`-question, the *Expected Answer Type* (EAT) is identified and replaced with '`ANSWER`' as a simplified form for the next step. For example, the EAT of the input question `Q1` would be '*director*.' In the next step, MEANS identifies medical entities using a *Conditional Random Field* (CRF) classifier and rules to map noun phrases to concepts. The next step is used to identify seven predefined semantic relations. The annotator is a hybrid approach based on a set of manually constructed patterns and a *Support Vector Machine* (SVM) classifier.

The strength of MEANS is that it can handle different types of questions, including questions with more than one expected answer type and more than one focus. As for most of the grammar-based NLIs, MEANS suffers from the restriction based on the handcrafted rules. The inclusion of ML reduces this problem, but ML itself needs huge training corpus to be usable. Furthermore, comparison (and also negation) is not taken into account.

5.4.5 AskNow

AskNow [14] uses a novel query characterization structure that is resilient to paraphrasing, called *Normalized Query Structure* (NQS), which is less sensitive to structural variation in the input question. The identification of the elements in the NQS is highly dependent on POS tags. For example, the input question `Q1` '*Who is the director of "Inglourious Basterds"?*' would be matched to the NQS template: `[Wh][R1][D][R2][I]`, where `[Wh]` is the question word '*Who*,' `[R1]` is the auxiliary relation '*is*,' `[D]` is the query desire class '*director*,' `[R2]` the relation '*of*' and `[I]` is the query input class '*Inglourious Basterds*.'

To translate the input question into SPARQL, AskNow first identifies the substructures using a POS tagger and named entity recognition. Then, it fits the substructures into their corresponding cells within the generic NQS templates. Afterward, the query type (set, boolean, ranking, count or property value) is identified based on desire and `wh`-type. In the next step, the query desire, query input and their relations will be matched to the KB. As an example, Spotlight can be used for the matching to DBpedia. During the matching process, AskNow uses WordNet synonyms and a BOA pattern library (bootstrapping).

The strength of AskNow, compared to the previous grammar-based systems is that the users are free to formulate their questions without restrictions. In addition, the NQS templates allow complex questions which, for example, can include subqueries. One weakness of AskNow is that it highly

depends on the right PoS tags and restricts the types of questions that can be asked.

5.4.6 SPARKLIS

SPARKLIS [19] is a guided query builder for SPARQL using natural language for better understanding. It guides the users during their query phrasing by giving the possibilities to search through concepts, entities and modifiers in natural language. It relies on the rules of SPARQL to ensure syntactically correct SPARQL queries all the time during the process. The interaction with the system makes the question formulation more constrained, slower and less spontaneous, but it provides guidance and safeness with intermediate answers and suggestions at each step. The translation process for SPARKLIS is reversed: it translates possible SPARQL queries into natural language such that the users can understand their choices.

The auto-completion is different from the previous systems (e.g., TR Discover and Ginseng): the interface displays three lists where the users can search for concepts, entities or modifiers. To ensure completeness relative to user input, SPARKLIS uses a cascade of three stages. The first stage is on client side, where a partial list of suggestion is filtered. The second stage is executed if the filtered list gets empty, then the suggestions is recomputed by sending the query, including the users filter, to the SPARQL endpoint. The last stage is triggered if the list is still empty, then, new queries are again sent to the SPARQL endpoint, using the full SPARQL query instead of partial results. Only a limited number of suggestions are computed and no ranking is applied, because of scalability issues.

The strength of SPARKLIS is also its weakness: the restricted guidance of the users during the query formulation process allows only syntactically correct questions, but at the same time, the users' freedom is reduced. Furthermore, the limited number of suggestions has the negative consequence that they may be incomplete and therefore making some queries unreachable.

5.4.7 GFMed

GFMed [41] is an NLI for biomedical linked data. It applies grammars manually built with *Grammatical Framework*[7] (GF). GF grammars are divided into *abstract* and *concrete* grammars. The *abstract* grammar defines the semantic model of the input language, and for GFMed, this is based on the biomedical domain. The *concrete* grammars define the syntax of the input language, which is English and SPARQL. Furthermore, GF supports multilingual applications and because of that Romanian is included as a second natural language in GFMed.

To translate the controlled natural language input into SPARQL, GFMed relies in a first step on the libraries of GF for syntax, morphological paradigms and coordination. GFMed covers basic elements of SPARQL to support term constraints, aggregates and negations. There is no support for property paths of length different from 1, optional graph pattern or assignment. Furthermore, only equality and regular expression operators are included.

The strength of GFMed is that it covers the basic elements of SPARQL. Furthermore, it introduces a second natural language besides of English for the users to ask questions. The weakness are the GF grammars which are domain-dependent and restrict the number of questions that can be asked by the users.

6 Evaluation

In this section, we first evaluate the 24 recently developed NLIs which were summarized before and systematically analyze if they are able to handle the ten sample questions of increasing complexity. This approach enables us to directly compare these systems which was previously not possible due to different datasets or evaluation measures used in the original papers. Afterward, we evaluate three commercial systems by asking them the same sample questions and analyzing their responses.

6.1 Evaluation of 24 recently developed NLIs

In this section, we provide a systematic analysis of the major NLIs. We categorized and analyzed the translation process of the 24 recently developed systems highlighted in Sect. 5 based on ten sample world questions with increasing complexity. Each system category, based on its technical approach, has its own strengths and weaknesses. There are also different aspects on which a system can focus (e.g., number of user interaction, ambiguity, efficiency, etc.,), which we do not take into account in this paper.

We evaluate the systems based on what is reported in the papers. If there is either an example of a similar question (e.g., '*Who is the president of the united states*?' (`Q1`) or a clear statement written in the paper (e.g., '*we can identify aggregations*' (`Q7`), we label those questions for the system with a checkmark (✓) in Table 3. If the question needs to be asked in a strict syntactical way (e.g., SODA needs the symbol '>' instead of '*higher than*') or the answer is partially correct (e.g., `Q4` returns a ordered list of movies instead of only one), it is labeled with a triangle (▲). If there is a clear statement that something is not implemented (e.g., ATHENA does not support negations), we label it with ✗. If we were not able to

[7] https://www.grammaticalframework.org/.

Table 3 Analysis of recently developed NLIs based on ten sample input questions

			Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	SQL	SPARQL
Keyword	SODA	*2012*	✓	▲	▲	✗	▲	✓	✗	✗	✗	✗	✓	✗
	NLP-Reduce	*2007*	✓	✗	?	✗	?	✓	✗	?	✗	✗	✗	✓
	Précis	*2008*	▲	✗	✗	✗	▲	✗	✗	▲	✗	✗	✓	✗
	QUICK	*2009*	✓	✗	✗	✗	✗	?	✗	?	✗	✗	✗	✓
	QUEST	*2013*	✓	?	?	?	?	✗	?	?	✗	✗	✓	✗
	SINA	*2015*	✓	?	?	✗	▲	✗	✗	?	✗	✗	✗	✓
	Aqqu	*2015*	✓	?	?	?	?	?	✗	?	✗	✗	✗	✓
Pattern	NLQ/A	*2017*	✓	✓	?	✓	✓	✓	✓	?	?	?	✗	✓
	QuestIO	*2008*	✓	✗	✗	✗	✓	?	✗	?	✗	✗	✗	✓
Parsing	ATHENA	*2016*	✓	✓	✓	✓	✓	✓	▲	✗	✓	✗	✓	✗
	Querix	*2006*	✓	?	?	✓	✓	?	?	?	?	?	✗	✓
	FREyA	*2010*	✓	?	?	✓	?	?	?	✗	?	?	✗	✓
	BELA	*2012*	✓	?	?	?	?	?	?	?	?	?	✗	✓
	USI Answers	*2013*	✓	✓	✓	?	✓	✓	▲	✓	?	?	✓	✓
	NaLIR (NaLIX)	*2014*	✓	✓	?	✓	✓	✗	✓	✓	✓	✓	✓	✗
	BioSmart	*2017*	✓	?	?	?	✓	✓	?	?	?	?	✓	✗
Grammar	TR Discover	*2015*	✓	?	?	✗	▲	?	✗	▲	?	?	✓	✓
	Ginseng	*2005*	✓	?	?	?	✓	?	?	?	?	?	✗	✓
	SQUALL	*2014*	▲	▲	▲	▲	▲	▲	▲	▲	▲	▲	✗	✓
	MEANS	*2015*	✓	✗	?	✗	?	✓	✗	✗	✗	✗	✗	✓
	AskNow	*2016*	✓	?	✓	?	✓	▲	?	?	✓	?	✗	✓
	SPARKLIS	*2017*	✓	✓	✓	▲	✓	?	▲	✓	✓	✓	✗	✓
	GFMed	*2017*	✓	✗	✗	✓	?	?	?	✓	?	?	✗	✓
Commercial	Google		✓	▲	✗	✓	✓	▲	▲	✓	✓	✓	?	?
	Siri		✓	✗	▲	✗	✗	✗	✗	✗	✗	✗	?	?
	IMDb		▲	✓	✓	✗	✓	✗	✗	✗	✗	✗	?	?

✓, can answer; ▲, strict syntax or partly answerable; ✗, cannot answer; ?, not documented

conclude, if a system can or cannot answer a question based on the paper, we labeled it with a question mark in Table 3.

In general, as shown in Table 3, we can say that keyword-based NLIs are the least powerful and can only answer simple questions like string filters (Q1). This limitation is based on the approach of these keyword-based systems: they expect just keywords (which are mostly filters) and the systems identify relationships between them. Therefore, they do not expect any complex questions like Q4 or Q7. Pattern-based NLIs are an extension of keyword-based systems in such a way that they have a dictionary with trigger words to answer more complex questions like aggregations (Q7). However, they cannot answer questions of higher difficulties, including subqueries (Q9/Q10). For example, the difficulty with questions including subqueries is to identify which part of the input question belongs to which subquery. Trigger words are not sufficient to identify the range of each subquery.

Parsing-based NLIs are able to answer these questions by using dependency or constituency trees (e.g., NaLIR). This helps to identify and group the input question in such a way that the different parts of the subqueries can be identified. Still, some of those systems struggle with the same problem as pattern-based NLIs: the systems are using trigger word dictionaries to identify certain types of questions like aggregations (e.g., ATHENA), which is not a general solution of the problem.

Grammar-based systems offer the possibility to guide the users during the formulation of their questions. Dynamically applying the rules during typing allows the systems to ensure that the input question is always translatable into formal language. The huge disadvantage of grammar-based systems is that they need handcrafted rules. There are NLIs which use a set of general rules and domain-specific rules (e.g., SQUALL). The general rules can be used for other domains and therefore increase the adaptability of the system. Other systems try to extract the rules directly from the ontology and thereby reduce the number of handcrafted rules (e.g., Ginseng).

We will now analyze how well the different systems can handle the ten sample questions. The first question is a basic filter question and can be solved by all NLIs as shown in Table 3. The last three questions are the most difficult ones and can only be answered by a few systems (completely by SQUALL, SPARKLIS and partially by others). Complex questions (e.g., aggregations) cannot be phrased with keywords only. There-

fore, the more complicated the users questions are, the more they will phrase them in grammatically correct sentences. In contrast, simple questions (e.g., string filters like `Q1`) can be easily asked with keywords. Both, Waltinger et al. [59] (USI Answer) and Lawrence and Riezler [33] (NLmaps) are describing this phenomenon and that the users prefer to ask questions with keywords if possible. They adapted their systems so that they can handle different forms of user input. Because of that Waltinger et al. [59] (USI Answer) point out that parse trees should only be used with caution. This is similar to the approach of Zheng et al. [69] (NLQ/A), who remark that NLP technologies are not worth the risk, because wrong interpretations in the processing leads to errors. Walter et al. [58] (BELA) propose a new approach of applying certain processing steps only if the question cannot be answered by using simpler mechanisms. This approach can be used to answer questions formulated as keywords or as complete sentences. Nevertheless, parse trees are useful to identify subqueries, but only in grammatically correct sentences (e.g., NaLIR). The identification of possible subqueries is necessary to answer questions like `Q9` and `Q10`.

Based on the sample questions, SQUALL, SPARKLIS, NaLIR and ATHENA are the systems that perform best (i.e., they can handle most of the question types). However, these systems still have some drawbacks. SQUALL requires that the user has knowledge of the RDF vocabulary. For example, if the users ask about all movies starring Brad Pitt, the question needs to be similar to '*All* `movies` *starring* `Brad_Pitt`.' SPARKLIS can answer all questions (if concepts are defined) but is based on a strict user interface where the users have to 'click their questions together' and cannot 'write freely.' In contrast to those two systems, NaLIR and ATHENA are systems where the user can write without restrictions during the process of phrasing the questions. However, NaLIR cannot handle concepts. Finally, ATHENA solves aggregations with trigger words which the users need to know. Moreover, ATHENA cannot solve multiple subqueries.

6.2 Evaluation of commercial systems

Increasingly, natural language interfaces are deployed for commercial usage. We decided to ask the ten sample questions to three commercial systems: Google,[8] Siri[9] and Internet Movie Database (IMDb).[10]

Based on the sample questions, Google is the best system in this category. An answer is assumed to be correct, if Google displays the answer in the featured snippet at the top of the results or in the knowledge panel on the side. For example, `Q1` is answered in the featured snippet. In contrast, for question `Q2` the featured snippet on top shows the top 250 drama movies, but the first result site contains the correct answer.

Siri can only answer simple `select` and `filter` questions and has some troubles to handle the year without a specific date in `Q3`. What is different to other systems is that if it cannot answer a question, Siri gives feedback to the user and tries to explain which type of questions can be answered. For most of the sample questions, we got the answer '*Sorry, I can't search what something is about. But I can search by title, actors or directors and categories like horror or action.*'

IMDb is a keyword-based system with the option of advanced searches[11] which are form-based. This system is not able to provide precise answers to `select` questions like `Q1` about the director of a given movie. However, the users can find the correct results by browsing the movie page.

7 Machine learning approaches for NLIs

In current research, more and more systems include machine learning (ML) in their translation process (e.g., MEANS) or ranking (e.g., Aqqu).

KBQA [10] learns templates as a kind of question representation. It supports binary factoid questions as well as complex questions which are composed of binary factoid questions. OQA [16] approaches to leverage both curated and extracted KBs, by mining millions of rules from an unlabeled question corpus and across multiple KBs.

AMUSE [22] uses ML to determine the most probable meaning of a question and can handle different languages at the same time. Xser [65] divides the translation task into a KB-independent and a KB-related task. In the KB-independent task, they are developing a *Directed Acyclic Graph* parser to capture the structure of the query intentions and trained it on human-labeled data.

A new promising avenue of research is to use deep learning techniques as the foundation for NLIDBs. The basic idea is to formulate the translation of natural language (NL) to SQL as an end-to-end machine translation problem [13,28,54]. The approach is often called neural semantic parsing [60]. In other words, translating from NL to SQL can be formulated as a supervised machine learning problem on pairs of natural language and SQL queries. In particular, machine translation can be modeled as a sequence-to-sequence problem where the input sequence is represented by the words (tokens) of the NL and the output sequence by the tokens of SQL. The main goal is given an input sequence of tokens, predict the output sequence based on observed patterns in the past.

The main advantage of machine learning-based approaches over traditional NLIDBs is that they support a richer lin-

[8] https://www.google.com/.

[9] https://www.apple.com/siri/.

[10] https://www.imdb.com.

[11] e.g., advanced title search https://www.imdb.com/search/title.

guistic variability in query expressions, and thus users can formulate queries with greater flexibility. However, one of the major challenges of supervised machine learning approaches is that they require a large training data set in order to achieve good accuracy on the translation task (see further details below).

The most commonly used approach for sequence-to-sequence modeling is based on recurrent neural networks (RNNs, [15]) with an input encoder and an output decoder. The encoder and decoder are implemented as bi-directional LSTMs (Long Short Term Memory) by Hochreiter and Schmidhuber [25]. However, before an NL can be encoded, the tokens need to be represented as a vector that in turn can be processed by the neural network. A widely used approach is to use word embeddings where the vocabulary of the NL is mapped to a high-dimensional vector [45]. In order to improve the accuracy of the translation, attention models are often applied [39].

One of the currently most advanced neural machine translation systems was introduced by Iyer et al. [26]. Their approach uses an encoder–decoder model with global attention similar to Luong et al. [39] and a bi-directional LSTM network to encode the input tokens. In order to improve the translation accuracy from NL to SQL compared to traditional machine translation approaches, the database schema is taken into account. Moreover, external paraphrases are used to increase the linguistic variance of the training data. In the first step of training the neural machine translation system, the process of generating training data is bootstrapped by manually handcrafted templates for the NL and SQL query pairs. In the next phase, the training set is extended by adding linguistic variations of the input questions and parts to the query are replaced with synonyms or paraphrases of the query. The advantage of this approach is that it is query language independent and could in principle also be used to translate from NL to SPARQL. However, the disadvantage is that a large, manually handcrafted a training set is necessary. The authors reported their results using two different data sets with a training data size of 600 and 4,473 utterances, respectively.

Zhong et al. [70] introduce a system called Seq2SQL. Their approach uses a deep neural network architecture with reinforcement learning to translate from NL to SQL. The authors released WikiSQL—a new data set based on Wikipedia consisting of 24,241 tables and 80,654 hand-annotated NL-SQL-pairs. However, their approach was only demonstrated to work on simple single-table queries without joins. SQLNet by Xu et al. [66] uses a more traditional machine translation approach without reinforcement learning. However, even though SQLNet shows better accuracy than Seq2SQL, the experiments are also based on the WikiSQL data set and it is not clear how this approach would handle join queries against more realistic database settings with multiple tables. Finally, Yavuz et al. [67] show further improvements over SQLNet by incorporating both the information about the database schema as well as the base data. The paper in particular focuses on the generation of WHERE-clauses, which the authors identified as major problem of the relative low accuracy of SQL queries generated by Seq2SQL and SQLNet.

DBPal by Basik et al. [1] overcomes shortcomings of manually labeling large training data sets by synthetically generating a training set that only requires minimal annotations in the database. Similar to Iyer et al. [26], DBPal uses the database schema and query templates to describe NL/SQL-pairs. Moreover, inspired by Wang et al. [60], DBPal augments an initial set of NL queries using a paraphrasing approach based on a paraphrase dictionary. The results show that on a single-table data set DPal performs better than the semantic parsing approach of Iyer et al. [26]. However, for a multi-table data set with join queries, DBPal performs worse. The reason is that the limited training set of 600 examples does not seem to generalize well for join queries. The authors attributed the good performance of the system introduced by Iyer et al. [26] to overfitting.

Soru et al. [53] use neural machine translation approach similar to Iyer et al. [26]. However, the major difference is that they translate natural language to SPARQL rather than to SQL. Moreover, they do not apply an attention mechanism. In a subsequent white paper, Hartmann et al. [23] present an approach to automatically generate a large set of training data consisting of 894,499 training examples based on set of 5000 natural language queries. The approach is very similar to the approach used by DBPal.

In general, these new approaches show promising results, but they have either only been demonstrated to work for single-table data sets or require large amounts training data. Hence, the practical usage in realistic database settings still needs to be shown.

Another interesting new trend is in the area of conversational systems such as Williams et al. [63], John et al. [29] that often apply neural machine translation techniques. However, a detailed discussion on these systems is beyond the scope of this paper.

8 Conclusions

In this paper, we summarized 24 recently developed natural language interfaces for databases. Each of them was evaluated using ten sample questions to show their strengths and weaknesses. Based on this evaluation, we identified the following lessons learned that are relevant for the development of NLIs.

Lesson 1—Use distinct mechanisms for handling simple versus complex questions Users like to pose questions differently depending on the complexity of the question.

Simple questions will often be asked with keywords, while complex questions are posed in grammatically correct sentences. For example, if users search for information about Brad Pitt, it is more likely that the users ask '*Brad Pitt*' as an input question and not '*Give me information about Brad Pitt*.' This lesson is highlighted in the paper by Waltinger et al. [59] (USI Answer) and related to the finding that casual users are more at ease at posing complex questions in natural language than in other formats [30]. Generally, pattern-based systems are solving this problem, but are often limited in the understanding of complex questions (i.e., subqueries). Walter et al. [58] (BELA) propose a layered system which could also be used to solve this problem of distinction. Grammar-based systems could be taught to support such non-natural language question patterns when included in the grammar. To the best of our knowledge, none of the systems we looked at supports this feature.

Lesson 2—Identifying subqueries still is a significant challenge for NLIs The identification of subqueries seems to be one of the most difficult problems for NLIs. The sample input questions `Q9` and `Q10` are two examples for questions composed of one and multiple subqueries, respectively. The most common NLP technology that is able to solve this problem is a parse tree. This can either be a general dependency or constituency parse tree provided, for example, by the Stanford Parser (e.g., NaLIR), or a parse tree self-learned with the rules of a grammar-based NLI (e.g., SQUALL). An alternative is the use of templates (e.g., AskNow). Li and Jagadish [35] (NaLIR) mention that the identification alone is not enough: after the identification of the subqueries, the necessary information needs to be propagated to each part of the subquery.

Lesson 3—Optimize the number of user interactions Natural language is ambiguous and even in a human-to-human conversation ambiguities occur, which are then clarified with questions. The same applies to NLIs: when an ambiguity occurs, the system needs to clarify with the user. This interaction should be optimized in such a way that the number of needed user interactions is minimized. To address this issue, Zenz et al. [68] (QUICK) and Zheng et al. [69] (NLQ/A) developed a minimization algorithm. The idea is to identify the ambiguity which has the most impact on the other ambiguities and clarify it first.

Lesson 4—Use grammar for guidance The biggest advantage of grammar-based NLIs is that they can use their grammar rules to guide the users while they are typing their questions. This improves the interaction between system and users in two ways: first, the system will understand each question the users ask; second, the users will learn how certain questions have to be asked to receive a good result. For the second part, Li et al. [38] (NaLIX) propose a solution by using a question historization and templates. This also helps the users understand what type of questions the system can understand and how the users need to ask.

Lesson 5—Use hybrid approach of traditional NLI systems and neural machine translation

New NLIs based on neural machine translation show promising results. However, the practical usage in realistic database settings still needs to be shown. Using a hybrid approach of traditional NLIs that are enhanced by neural machine translation might be a good approach for the future. Traditional approaches would guarantee better accuracy while neural machine translation approaches would increase the robustness to language variability.

In summary, our evaluation of various systems against ten sample questions shows that NLIs have made significant progress over the last decade. However, our lessons also show that more research is required to increase the expressive power of NLIs. This paper gives a guide for researchers and practitioners highlighting the strengths and weaknesses of current approaches as well as helps them design and implement NLIs that are not only of theoretical value but have impact in industry.

Acknowledgements The work was supported by Hasler Foundation under Grant Number 17040.

A SQL Representation

`Q1:`

```
SELECT DISTINCT p.*
FROM movie m, person p,
 directing d
WHERE m.id = d.movieId
  AND person.id = d.directorId
  AND m.title = 'Inglourious
 Basterds'
```

`Q2:`

```
SELECT * FROM movie m WHERE
m.rating > 9
```

`Q3:`

```
SELECT DISTINCT m.*
FROM movie m, starring s,
person p
WHERE m.id = s.movieId
  AND s.actorId = p.id
```

```
  AND p.name = 'Brad Pitt'
  AND m.releaseDate >=
'2000-01-01'
  AND m.releaseDate <=
 '2010-12-31'
```

Q4:

```
SELECT DISTINCT m.*
FROM movie m, gross g
WHERE m.id = g.movieId
  AND g.gross = (
    SELECT MAX(gross) FROM gross
  )
```

Q5:

```
SELECT DISTINCT m.*
FROM movie m, genre g
WHERE m.id = g.movieId
    AND (
      g.genre = 'drama'
      OR g.genre = 'comedy'
    )
```

Q6:

```
SELECT * FROM movie m WHERE
m.rating >= 8
```

Q7:

```
SELECT DISTINCT m.*
FROM movie m, genre g, (
    SELECT g.genre, max(m.rating)
as maxRating
    FROM movie m, genre g
    WHERE m.id = g.moveId
    GROUP BY g.genre
  ) as maxMovie
WHERE m.id = g.movieId
  AND m.rating = maxMovie.
maxRating
  AND g.genre = maxMovie.genre
```

Q8:

```
SELECT DISTINCT m.*
FROM movie m, genre g
WHERE m.id = g.movieId
    AND originalLang != 'jp'
    AND g.genre = 'horror'
```

Q9:

```
SELECT *
FROM movie m1
WHERE m1.rating > (
  SELECT m2.rating
  FROM movie m2
  WHERE m2.title = 'Sin City'
)
```

Q10:

```
SELECT DISTINCT m1.*
FROM genre g1, movie m1
WHERE m1.id = g1.movieId
  AND NOT EXISTS (
    SELECT ''
    FROM movie m2
    WHERE m2.title = 'Sin City'
      AND NOT EXISTS (
        SELECT  ''
        FROM genre g2
        WHERE g2.movieId = m1.id
          AND g2.id = g1.id
      )
  )
```

References

1. Basik, F., Hättasch, B., Ilkhechi, A., Usta, A., Ramaswamy, S., Utama, P., Weir, N., Binnig, C., Cetintemel, U.: DBPal: A learned NL-interface for databases. In: Proceedings of the 2018 International Conference on Management of Data, pp. 1765–1768. ACM (2018)
2. Bast, H., Haussmann, E.: More accurate question answering on freebase. In: Proceedings of 24th ACM International Conference on Information and Knowledge Management—CIKM '15, pp. 1431–1440 (2015)
3. Ben Abacha, A., Zweigenbaum, P.: MEANS: a medical question-answering system combining NLP techniques and semantic Web technologies. Inf. Process. Manag. **51**(5), 570–594 (2015)
4. Bergamaschi, S., Guerra, F., Interlandi, M., Trillo-Lado, R., Velegrakis, Y.: Combining user and database perspective for solving keyword queries over relational databases. Inf. Syst. **55**, 1–19 (2016)
5. Bernstein, A., Kaufmann, E., Kaiser, C.: Querying the semantic web with ginseng: a guided input natural language search engine. In: 15th Work Information Technology System, Las Vegas, NV, pp. 112–126 (2005)
6. Blunschi, L., Jossen, C., Kossmann, D., Mori, M., Stockinger, K.: SODA: Generating SQL for Business Users. Proc VLDB Endow **5**(10), 932–943 (2012)
7. Bonifati, A., Martens, W., Timm, T.: An analytical study of large SPARQL query logs. Proc. VLDB Endow. **11**(2), 149–161 (2017). arXiv:1708.00363
8. Bowen, P., Chang, C., Rohde, F.: Non-length based query challenges: an initial taxonomy. In: 14th Annual Workshop on Information Technology and Systems, WITS, pp. 74–79 (2004)
9. Codd, E.F.: Seven steps to rendezvous with the casual user. In: IFIP Working Conference Database Management, pp. 179–200 (1974)
10. Cui, W., Xiao, Y., Wang, H., Song, Y., Sw, Hwang, Wang, W.: KBQA: learning question answering over QA corpora and knowledge bases. Proc. VLDB Endow. **10**(5), 565–576 (2017)
11. Damljanovic, D., Tablan, V., Bontcheva, K., Court, R., Street, P.: A text-based query interface to OWL ontologies. In: Proceedings of

International Conference on Language Resources and Evaluation (LREC 2008), pp. 205–212 (2008)
12. Damljanovic, D., Agatonovic, M., Cunningham, H.: Natural Language Interfaces to Ontologies: Combining Syntactic Analysis and Ontology-Based Lookup Through the User Interaction. Springer, Berlin (2010)
13. Dong, L., Lapata, M.: Language to logical form with neural attention. arXiv preprint arXiv:1601.01280
14. Dubey, M., Dasgupta, S., Sharma, A., Höffner, K., Lehmann, J.: AskNow: a framework for natural language query formalization in SPARQL. In: International Semantic Web Conference, vol. 9678, pp. 300–316. Springer (2016)
15. Elman, J.L.: Finding structure in time. Cognit. Sci. **14**(2), 179–211 (1990)
16. Fader, A., Zettlemoyer, L., Etzioni, O.: Open question answering over curated and extracted knowledge bases. In: Proceedings of 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining—KDD '14, pp. 1156–1165 (2014)
17. Ferré, S.: SQUALL: a High-Level Language for Querying and Updating the Semantic Web. Technical report (2011)
18. Ferré, S.: SQUALL: the expressiveness of SPARQL 1.1 made available as a controlled natural language. Data Knowl. Eng. **94**, 163–188 (2014)
19. Ferré, S.: SPARKLIS: an expressive query builder for SPARQL endpoints with guidance in natural language. Open J Semant Web, Res Online Publ 0 (2017)
20. Gautam, A., Kumar, S., Mittal, S.: Survey on natural language database interfaces. Int. J. Adv. Res. Comput. Sci. **8**(5), 469–473 (2017)
21. Green, B., Wolf, A., Chomsky, C., Laughery, K.: Baseball: an automatic question answerer. In: Proceedings of Western Joint Computer Conference, pp. 219–224 (1961). arXiv:1011.1669v3
22. Hakimov, S., Jebbara, S., Cimiano, P.: AMUSE: multilingual semantic parsing for question answering over linked data. In: Proceedings of 16th International Semantic Web Conference (ISWC 2017), pp. 1–16 (2017)
23. Hartmann, A.K., Tommaso, Marx E., Moussallem, D., Publio, G., Valdestilhas, A., Esteves, D., Neto, C.B.: Generating a large dataset for neural question answering over the dbpedia knowledge base. ResearchGate (2018)
24. Hendrix, G.G., Sacerdoti, E.D., Sagalowicz, D., Slocum, J.: Developing a natural language interface to complex data. ACM Trans. Database Syst. (TODS) **3**(2), 105–147 (1978)
25. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997)
26. Iyer, S., Konstas, I., Cheung, A., Krishnamurthy, J., Zettlemoyer, L.: Learning a neural semantic parser from user feedback. In: 55th Annual Meeting of the Association for Computational Linguistics (2017)
27. Jamil, H.M.: Knowledge Rich natural language queries over structured biological databases. In: Proceedings of 8th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics, pp. 352–361 (2017). arXiv:1703.10692v1
28. Jia, R., Liang, P.: Data recombination for neural semantic parsing (2016). arXiv preprint arXiv:1606.03622
29. John, R.J.L., Potti, N., Patel, J.M.: Ava: From data to insights through conversations. In: CIDR (2017)
30. Kaufmann, E., Bernstein, A.: Evaluating the usability of natural language query languages and interfaces to Semantic Web knowledge bases. J. Web Semant. **8**(4), 377–393 (2010)
31. Kaufmann, E., Bernstein, A., Zumstein, R.: Querix: a natural language interface to query ontologies based on clarification dialogs. In: ISWC (November), pp. 980–981 (2006)
32. Kaufmann, E., Bernstein, A., Fischer, L.: NLP-reduce: a naive but domain independent natural language interface for querying ontologies. In: 4th European Semantic Web Conference ESWC, pp. 1–2 (2007)
33. Lawrence, C., Riezler, S.: NLmaps: a natural language interface to query OpenStreetMap. In: COLING (Demos), pp. 6–10 (2016)
34. Lehmann, J., Isele, R., Jakob, M., Jentzsch, A., Kontokostas, D., Mendes, P.N., Hellmann, S., Morsey, M., Van Kleef, P., Auer, S., Bizer, C.: DBpedia—a large-scale, multilingual knowledge base extracted from wikipedia. Semant. Web **1**, 1–5 (2012)
35. Li, F., Jagadish, H.V.: Constructing an interactive natural language interface for relational databases. Proc. VLDB Endow. **8**(1), 73–84 (2014)
36. Li, F., Jagadish, H.V.: NaLIR: an interactive natural language interface for querying relational databases. In: Proc 2014 ACM SIGMOD International Conference on Management of Data, pp. 709–712 (2014)
37. Li, Y., Rafiei, D.: Natural language data management and interfaces. In: Proceedings of the 2017 ACM International Conference on Management of Data—SIGMOD '17, pp. 1765–1770. ACM (2017)
38. Li, Y., Yang, H., Jagadish, H.: Nalix: a generic natural language search environment for XML data. ACM Trans. Database Syst. (TODS) **32**(4), 30 (2007)
39. Luong, T., Pham, H., Manning, C.D.: Effective approaches to attention-based neural machine translation. In: Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, pp. 1412–1421 (2015)
40. Manning, C., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S., McClosky, D.: The Stanford CoreNLP natural language processing toolkit. In: Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations, pp. 55–60 (2014). arXiv:1011.1669v3
41. Marginean, A.: Question answering over biomedical linked data with Grammatical Framework. Semant. Web **8**(4), 565–580 (2017)
42. Miller, G.A.: WordNet: a lexical database for English. Commun. ACM **38**(11), 39–41 (1995)
43. Mishra, A., Jain, S.K.: A survey on question answering systems with classification. J. King Saud Univ. Inf. Sci. **28**(3), 345–361 (2016)
44. Nihalani, N., Silakari, S., Motwani, M.: Natural language interface for database: a brief review. IJCSI Int. J. Comput. Sci. Issues **8**(2), 600–608 (2011)
45. Pennington, J., Socher, R., Manning, C.: Glove: Global vectors for word representation. In: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), pp. 1532–1543 (2014)
46. Porter, M.F.: An algorithm for suffix stripping. Program **14**(3), 130–137 (1980)
47. Sacerdoti, E.D.: Language access to distributed data with error recovery. In: Proceedings of IJCAI-77, pp. 196–202 (1977)
48. Saha, D., Floratou, A., Sankaranarayanan, K., Farooq Minhas, U., Mittal, A.R., Ozcan, F.: ATHENA: an ontology-driven system for natural language querying over relational data stores. Proc. VLDB Endow. **9**(12), 1209–1220 (2016)
49. Shafique, U., Qaiser, H.: A comprehensive study of natural language interface to database. Int. J. Innov. Sci. Res. **9**(2), 297–306 (2014)
50. Shekarpour, S., Marx, E., Ngonga Ngomo, A.C., Auer, S.: SINA: semantic interpretation of user queries for question answering on interlinked data. Web Semant. Sci. Serv. Agents World Wide Web **30**, 39–51 (2015)
51. Simitsis, A., Koutrika, G., Ioannidis, Y.: Précis: from unstructured keywords as queries to structured databases as answers. VLDB J. **17**(1), 117–149 (2008)
52. Song, D., Schilder, F., Smiley, C., Brew, C., Zielund, T., Bretz, H., Martin, R., Dale, C., Duprey, J., Miller, T., Harrison, J.: TR discover: a natural language interface for querying and analyz-

ing interlinked datasets. In: Semantic Web-ISWC 2015, pp. 21–37 (2015)
53. Soru, T., Marx, E., Moussallem, D., Publio, G., Valdestilhas, A., Esteves, D., Neto, C.B.: Sparql as a foreign language. In: 13th International Conference on Semantic Systems (2017)
54. Sutskever, I., Vinyals, O., Le, Q.: Sequence to sequence learning with neural networks. Adv. Neural Inf. Process. Syst. **27**, 3104–3112 (2014)
55. Szalay, A.S., Gray, J., Thakar, A.R., Kunszt, P.Z., Malik, T., Raddick, J., Stoughton, C., vandenBerg, J.: The SDSS skyserver: public access to the sloan digital sky server data. In: SIGMOD 202 (2002)
56. Tang, L., Mooney, R.: Using multiple clause constructors in inductive logic programming for semantic parsing. In: Machine Learning ECML 2001(September), pp. 466–477 (2001)
57. Thompson, B., Thompson, F.: Introducing ASK, a simple knowledgeable system. In: Proceedings of first Conference on Applied Natural Language Process, pp. 17–24 (1983)
58. Walter, S., Unger, C., Cimiano, P., Bär, D.: Evaluation of a layered approach to question answering over linked data. In: Semant Web-ISWC 2012, pp. 362–374 (2012)
59. Waltinger, U., Tecuci, D., Olteanu, M.: USI answers: natural language question answering over (semi-) structured industry data. In: Association for the Advancement of Artificial Intelligence, pp. 1471–1478 (2013)
60. Wang, Y., Berant, J., Liang, P.: Building a semantic parser overnight. In: Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), vol. 1, pp. 1332–1342 (2015)
61. Warren, D.H., Pereira, F.C.: An efficient for interpreting easily adaptable system natural language queries. Am. J. Comput. Linguist. **8**(3–4), 110–122 (1982)
62. Weischedel, R.: A hybrid approach to representation in the JANUS natural language processor. In: Proceedings of 27th Annual Meeting of the Association for Computational Linguistics, pp. 193–202 (1989)
63. Williams, J.D., Kamal, E., Ashour, M., Amr, H., Miller, J., Zweig, G.: Fast and easy language understanding for dialog systems with microsoft language understanding intelligent service (LUIS). In: Proceedings of the 16th Annual Meeting of the Special Interest Group on Discourse and Dialogue, pp. 159–161 (2015)
64. Woods, W.A.: Progress in natural language understanding: an application to lunar geology. In: Proceedings of National Computer Conference and Exposition. AFIPS '73, pp. 441–450 (1973)
65. Xu, K., Zhang, S., Feng, Y., Zhao, D.: Answering natural language questions via phrasal semantic parsing. In: Zong, C., Nie, JY., Zhao, D., Feng, Y. (eds.) Natural Language Processing and Chinese Computing. Communications in Computer and Information Science, vol, 496, pp. 333–344. Springer, Berlin, Heidelberg (2014)
66. Xu, X., Liu, C., Song, D.: SQLNet: Generating structured queries from natural language without reinforcement learning (2017). arXiv preprint arXiv:1711.04436
67. Yavuz, S., Gur, I., Su, Y., Yan, X.: What it takes to achieve 100% condition accuracy on wikisql. In: Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, pp. 1702–1711 (2018)
68. Zenz, G., Zhou, X., Minack, E., Siberski, W., Nejdl, W.: From keywords to semantic queries-Incremental query construction on the semantic web. J. Web Semant. **7**(3), 166–176 (2009)
69. Zheng, W., Cheng, H., Zou, L., Yu, J.X., Zhao, K.: Natural language question/answering: let users talk with the knowledge graph. In: Proceedings of 2017 ACM Conference on Information and Knowledge Management, pp. 217–226. ACM (2017)
70. Zhong, V., Xiong, C., Socher, R.: Seq2SQL: Generating structured queries from natural language using reinforcement learning (2017). arXiv preprint arXiv:1709.00103